Politechnika Warszawska

WYDZIAŁ ELEKTRYCZNY

Instytut Elektrotechniki Teoretycznej
i Systemów Informacyjno-Pomiarowych
Zakład Elektrotechniki Teoretycznej
i Informatyki Stosowanej

Praca dyplomowa magisterska

na kierunku Informatyka
w specjalności Inżynieria oprogramowania

Projektowanie i implementacja elementów składowych Domain-Driven Design w języku Java

Mateusz Rasiński
nr albumu 229357

promotor
dr inż. Tomasz Leś

WARSZAWA 2017

PROJEKTOWANIE I IMPLEMENTACJA ELEMENTÓW SKŁADOWYCH DOMAIN-DRIVEN DESIGN W JĘZYKU JAVA

Streszczenie

Praca prezentuje właściwe sposoby projektowania i implementacji elementów składowych Domain-Driven Design w języku Java. Domain-Driven Design jest podejściem do wytwarzania oprogramowania, które cieszy się coraz większą popularnością w środowisku programistów aplikacji internetowych. Elementy składowe DDD to jego fundamentalne części, więc właściwe posługiwanie się nimi jest niezwykle istotne do robienia dobrego użytku z tego podejścia. Każdy z elementów został zanalizowany pod kątem jego prawidłowego projektu i implementacji. Ponadto, przedstawiono wady i zalety różnych podejść do ich tworzenia i korzystania z nich. Zostały także stworzone szczegółowe wytyczne ich projektowania wraz z ich przykładowymi implementacjami. Na szczególną uwagę zasługują nowe wnioski lub nowe proponowane zasady, którymi należy się kierować projektując i implementując elementy składowe DDD. Niniejsza praca może mieć zastosowanie jako przewodnik po elementach składowych DDD i stanowić wzór, w jaki sposób powinny one być projektowane i implementowane.

Słowa kluczowe: DDD, elementy składowe, Java, wytwarzanie oprogramowania

DESIGN AND IMPLEMENTATION OF DOMAIN-DRIVEN DESIGN BUILDING BLOCKS IN JAVA

Abstract

This thesis presents the proper ways of designing and implementing the Domain-Driven Design building blocks in Java. Domain-Driven Design is an approach to software development enjoying increasing popularity amongst web application developers. DDD building blocks are its fundamental parts, therefore the appropriate handling of them is essential to make good use of this approach. Each block has been analyzed in terms of its proper design and implementation. Furthermore, there have been presented the advantages and disadvantages of several approaches to creating and using them. There have also been introduced detailed guidelines on designing them along with their exemplary implementations. Particular attention should be given to new findings and newly proposed principles that should drive the design and implementation of the DDD building blocks. This thesis may be used as a guide to the DDD building blocks and serve as a good reference point for designing and implementing them correctly.

Keywords: DDD, building blocks, Java, software development

WARSZAWA, 4 grudnia 2017

POLITECHNIKA WARSZAWSKA
WYDZIAŁ ELEKTRYCZNY

OŚWIADCZENIE

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa magisterska pt. Projektowanie i implementacja elementów składowych Domain-Driven Design w języku Java:

- została napisana przeze mnie samodzielnie,
- nie narusza niczyich praw autorskich,
- nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam, że przedłożona do obrony praca dyplomowa nie była wcześniej podstawą postępowania związanego z uzyskaniem dyplomu lub tytułu zawodowego w uczelni wyższej. Jestem świadom, że praca zawiera również rezultaty stanowiące własności intelektualne Politechniki Warszawskiej, które nie mogą być udostępniane innym osobom i instytucjom bez zgody Władz Wydziału Elektrycznego.

Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

Mateusz Rasiński

.....

Mamie i Tacie

Spis treści

Rozdział 1

Wstęp

Tworzenie aplikacji internetowych polega na realizacji wymagań klienta biznesowego poprzez interaktywny system informatyczny. Przestrzeń problemu — biznes klienta — jest odpowiednio mapowana na przestrzeń rozwiązania — system informatyczny. Biznes klienta jest systemem, w którego skład wchodzą przypadki użycia oraz logika biznesowa, które są tłumaczone na struktury danych i procedury występujące w kodzie aplikacji. Jednym z podejść realizacji systemu informatycznego, który kładzie nacisk na dobre odwzorowanie logiki biznesowej jest Domain-Driven Design.

To podejście wprowadził już w 2004 roku Eric Evans w swojej książce zatytułowanej „Domain-Driven Design: Tackling Complexity in the Heart of Software" [2004]. Opisuje w niej holistyczne podejście do wytwarzania oprogramowania nakierowanego na modelowanie dziedziny problemu. Pojęcia wprowadzone w tej książce są prezentowane na wysokim poziomie abstrakcji, często niezależnie od konkretnego sposobu ich implementacji. Prawie dekadę później, w 2013 roku, zostaje wydana kolejna bardzo istotna książka dla obszaru DDD: „Implementing Domain-Driven Design" autorstwa Vaughna Vernona [2013]. Przykłady implementacji elementów składowych DDD są tu przedstawione zdecydowanie jaśniej niż w książce Evansa, jednak niektórym konceptom projektowym lub implementacyjnym autor poświęca mało miejsca lub nie poświęca go w ogóle. Poza tymi dwoma „bibliami DDD", jak nazywane są te książki w społeczeństwie DDD, nie istnieje wiele kompleksowych opracowań w jaki sposób powinny być projektowane i implementowane pojęcia wchodzące w skład DDD, w tym jego elementy składowe.

1.1 Cel i zakres pracy

W związku z powyższym, celem niniejszej pracy magisterskiej jest szczegółowe opracowanie sposobów na projektowanie i implementację elementów składowych Domain-Driven Design w konkretnym języku programowania — języku Java.

Zakres pracy ogranicza się do analizy sześciu podstawowych elementów składowych DDD: encji, obiektów wartości, agregatów, fabryk, repozytoriów oraz usług domenowych. W zakresie nie znajdują się pojęcia, które są czasem również zaliczane do elementów składowych, np. zdarzenia domenowe, polityki czy specyfikacje. Praca nie dotyczy również podejść strategicznych, czy architektury aplikacji proponowanej przez DDD.

1.2 Układ pracy

W rozdziale 2 („Domain-Driven Design") znajduje się krótka prezentacja podejścia Domain-Driven Design. Sekcja 2.1 zawiera charakterystykę głównych cech DDD, natomiast sekcja 2.2 ukazuje powiązania podejścia DDD z paradygmatem programowania obiektowego.

Rozdziały 3– 8 omawiają kolejne elementy składowe DDD. Każdy z tych rozdziałów rozpoczyna się krótkim opisem danego elementu składowego, następnie omawiane są jego charakterystyczne założenia projektowe lub problemy na które należy zwrócić szczególną uwagę. Rozdziały kończą się zazwyczaj na proponowanej implementacji z dokładnym jej omówieniem i komentarzem.

Rozdział 3 („Encje") został nietypowo podzielony na dwie duże części. Pierwsza część (3.1 „Unikalna tożsamość") traktuje o najistotniejszej cesze encji — jej tożsamości. Część druga (3.2 „Interakcje") prezentuje sposoby pracy z encjami poprzez wysyłanie i odbieranie komunikatów.

W rozdziale 4 („Obiekty wartości") oprócz opisania typowego przykładu obiektu wartości, w sekcji 4.1 zaproponowane zostały dwa inne przykłady zastosowań obiektów wartości. W sekcji 4.2 („Implementacja") znalazły się najbardziej generyczne przykłady implementacji w całej pracy. Są to implementacje referencyjne klas niemutowalnych i metod ich porównujących w języku Java, które można traktować jako gotowy wzór do zastosowania przy tworzeniu konkretnych implementacji takich klas i metod.

Rozdział 5 („Agregaty") czerpie dużo z poprzednich rozdziałów 3 i 4, bardziej skupiając się na właściwych przesłankach, które należy rozważyć przy projektowaniu agregatów, aniżeli na konkretnej implementacji. Sekcja 5.3 zawiera jednakże również i takową wraz z odpowiednimi komentarzami.

W rozdziale 6 („Fabryki"), w sekcji 6.1 omawiane są cztery rodzaje fabryk, natomiast w sekcji 6.2 prezentowana i komentowana jest przykładowa implementacja wybranych dwóch głównych spośród nich.

Rozdział 7 („Repozytoria") opisuje kolejny element składowy DDD. W sekcji 7.1 oceniana jest współpraca całego kodu aplikacji z tym elementem. Sekcja 7.2 zawiera opis dwóch podejść do odpytywania repozytorium. Sekcja 7.3 jest poświęcona kompleksowej analizie implementacji tego elementu.

Rozdział 8 („Usługi domenowe") przedstawia motywacje tworzenia usług domenowych i sposoby ich implementacji. Sekcja 8.1 zawiera krótki opis klas użytkowych jako usług domenowych. Sekcja 8.2 poświęcona jest ilustracji roli usługi domenowej jako orkiestrującej działania wielu agregatów. W sekcji tej przeprowadzono porównanie dwóch podejść rozwiązywania podobnej klasy problemów — z udziałem usługi domenowej i bez. Sekcja 8.3 uzupełnia rozważania o dodatkowe uwagi.

Rozdział 9 („Podsumowanie") stanowi podsumowanie pracy — streszcza powstałe w jej wyniku wnioski i twórcze innowacje, a także wskazuje perspektywy jej rozwoju.

Rozdział 2

Domain-Driven Design

Tworzenie aplikacji internetowych polega na realizacji wymagań biznesowych klienta biznesowego poprzez interaktywny system informatyczny. Przestrzeń problemu — biznes klienta — jest odpowiednio mapowana na przestrzeń rozwiązania — system informatyczny. Biznes klienta jest systemem, w którego skład wchodzą przypadki użycia oraz logika biznesowa — cechy i zachowania modeli biznesowych. Jednym z podejść do realizacji systemu informatycznego, który kładzie nacisk na dobre odwzorowanie logiki biznesowej jest Domain-Driven Design.

Domain-Driven Design (DDD, pol. projektowanie sterowane dziedziną) — to podejście do wytwarzania oprogramowania dla złożonych potrzeb poprzez dogłębne powiązanie implementacji z ewoluującym modelem najistotniejszych pojęć biznesowych [Evans, 2007]. DDD skupia się na modelowaniu biznesowym, nie na konkretnych sposobach implementacji. Wprowadzając koncepcje na różnym poziomie abstrakcji nie daje gotowych sposobów na ich specyficzną implementację zależną m.in. od języka programowania.

2.1 Charakterystyka

DDD adresuje problemy wynikające ze złożoności systemów informatycznych, zarówno esencjonalnej jak i przypadkowej [Sobótka, 2011]. Główne idee wprowadzone przez DDD to:

- język wszechobecny;
- rozdzielenie warstwy logiki na logikę aplikacyjną oraz biznesową;
- skupienie się na domenie (jej modelu);
- projektowanie strategiczne.

2.1.1 Język wszechobecny

Osoba znająca działanie biznesu klienta (w DDD nazywana ekspertem domenowym) posługuje się często żargonem biznesowym, nie znając żargonu technicznego. Analogicznie, twórcy systemów informatycznych posługują się swoim żargonem, nie posiadając dostatecznej wiedzy o żargonie i pojęciach występujących w dziedzinie klienta. Programiści muszą sobie jakoś tłumaczyć pojęcia biznesowe otrzymywane od klienta na występujące w kodzie nazwy klas i metod. W rozmowach między sobą mogą przez to powstawać nieporozumienia przez posługiwanie się czasem pojęciem biznesowym, a czasem technicznym. Ponadto, omawiając dane koncepty zaimplementowane w kodzie z klientem muszą dokonać odwrotnego tłumaczenia kodu na pojęcia biznesowe, co może powodować inne nieporozumienia tym razem z ekspertem domenowym. „Konieczność tłumaczenia osłabia komunikację, a także cały proces przetwarzania wiedzy" [Evans, 2004].

Z tych powodów, DDD wprowadza pojęcie języka wszechobecnego (ang. ubiquitous language), to jest takiego języka, którym posługują się wszyscy członkowie zespołu projektowego niezależnie od ich roli w projekcie. Jednoznaczność nazewnicza minimalizuje liczbę wystąpień nieporozumień, a także często identyfikuje i doprecyzowuje pojęcia i procesy biznesowe oraz techniczne, które wcześniej nie były brane pod uwagę zarówno przez programistów, jak i ekspertów domenowych. Wspólny język wymaga również staranności i ostrożności przy powoływaniu do życia nowych terminów, co czyni projekt bardziej przemyślanym.

2.1.2 Architektura aplikacji

Typowa aplikacja internetowa w języku Java składa się z trzech warstw: warstwy dostępu do danych, warstwy prezentacji oraz warstwy logiki. Ta ostatnia w DDD jest rozdzielona na dwie: warstwę logiki aplikacji i domeny. Decyzja ta jest spowodowana próbą destylacji wiedzy o mechanizmach biznesowych od ich wykorzystania. W kodzie aplikacji z jedną warstwą logiki te dwie rzeczy są ze sobą mocno powiązane, co powoduje powstawanie wielolinijkowych klas-serwisów, które charakteryzują się wysoką zależnością (ang. coupling [Constantine et al., 1974]).

Logika aplikacji składa się z serwisów aplikacyjnych, które są odpowiedzialne głównie za orkiestrację obiektów domenowych (żyjących w warstwie domenowej) w celu realizacji kolejnych kroków historyjek użytkownika (ang. User Stories) lub przypadków użycia (ang. Use Cases). Kod w tej warstwie jest proceduralny. Inne zagadnienia, za które odpowiedzialna jest ta warstwa, to zagadnienia techniczne dotyczące całej aplikacji, takie jak: transakcje, bez-

pieczeństwo, logowanie.

Logika domenowa (inaczej: biznesowa) to warstwa zawierająca wiedzę o tym, w jaki sposób funkcjonują obiekty biznesowe. Skupia w sobie model zachowań i reguł, które są niezbędne do prawidłowego działania systemu. Ta warstwa stanowi jądro systemu i jest niezależna od jakichkolwiek zewnętrznych technikaliów. To tu najbardziej przydaje się wiedza eksperta domenowego, dlatego ważne jest, żeby korzystać tu mocno z języka wszechobecnego. Kod w tej warstwie powinien być jak najbardziej obiektowy, żeby jak najlepiej oddać realia biznesowe.

2.1.3 Model

Model został uznany za sedno wg. DDD, gdyż większość problemów w projektach informatycznych dotyczy właśnie odpowiedniego odwzorowania świata fizycznego (biznesowego) na model przetwarzany przez system komputerowy. Do właściwego zobrazowania różnych pojęć biznesowych DDD definiuje szereg elementów składowych (ang. building blocks, dosł. klocki budulcowe).

Elementy składowe to fundamentalne elementy, z których można zbudować system, który odzwierciedla domenę biznesową klienta. Każdy z nich określa rodzaj klasy odpowiedzialnej za pełnioną rolę. Różnią się one zachowaniami oraz budową wewnętrzną. Elementy są w różnych relacjach między sobą, często prowadząc interakcje. Wyróżnia się następujące elementy składowe:

- encje;
- obiekty wartości;
- agregaty;
- fabryki;
- repozytoria;
- usługi domenowe.

2.1.4 Projektowanie strategiczne

DDD zajmuje się także wysoko poziomowym projektowaniem systemów poprzez odpowiednie utrzymywanie integralności modelu oraz destylację domen. Techniki projektowania strategicznego nie skupiają się na kodzie aplikacji, ale na architekturze wyższego rzędu — jak budować duże systemy, jak dzielić je na mniejsze części i jak te części ze sobą komunikować.

Utrzymywanie integralności modelu

Jednym z kluczowych pojęć wykorzystywanych przy projektowaniu strategicznym jest kontekst związany (ang. bounded context). Bazuje on na fakcie, że tak samo nazywany obiekt biznesowy w jednym kontekście może mieć zupełnie inne znaczenie w innym kontekście. W celu poprawnego zamodelowania systemu należy go rozdzielić na dwa konteksty związane. Konteksty związane można często utożsamiać z osobną aplikacją, którą może tworzyć osobny zespół. Komunikacja pomiędzy kontekstami może być prowadzona na różne sposoby. Do poprawnej komunikacji pomiędzy kontekstami jest wymagane zdefiniowanie mapy kontekstów (ang. context map), która służy do poprawnego tłumaczenia pojęć z jednego kontekstu na pojęcia występujące w drugim.

Destylacja domen

Destylacja domen (dziedzin) to technika poprzez którą można dodatkowo rozdzielić system na odpowiednie domeny, do których realizacji można podejść w inny sposób. DDD wyróżnia trzy rodzaje domen:

- główną (ang. core);
- pomocniczą (ang. supporting);
- ogólną (ang. generic).

Domena główna to najważniejszy komponent systemu dający największą wartość biznesową, która może stanowić przewagę konkurencyjną. Poświęca się jej najwięcej uwagi, stosując najlepsze techniki i angażując najlepszych ludzi. To tu należy skupić się nad poprawnym zastosowaniem wzorców taktycznych DDD, takich jak elementy składowe.

Domena pomocnicza to istotne komponenty systemu, ale nie kluczowe. Te komponenty wspierają pracę domeny głównej. Jakość kodu w tej domenie może być niższa. Implementacją tej domeny mogą się zająć np. pracownicy z firmy zewnętrznej.

Domena ogólna to domena, która nie jest zależna od systemu, który jest projektowany, a bardziej od użytych technologii. Do domen ogólnych można zaliczyć wszystkie sterowniki do bazy danych, kolejek lub innych narzędzi zewnętrznych. Domeny ogólne często mogą być produktami innych firm lub bibliotekami otwartoźródłowymi.

2.2 Związki z programowaniem obiektowym

DDD w naturalny sposób buduje na paradygmacie programowania obiektowego. W tym paradygmacie procedury i struktury danych są blisko związane w bytach zwanych obiektami. Obiekty mają na celu odwzorowywać pewien model świata rzeczywistego, tak jak w DDD, gdzie obiekty mają być modelem domeny biznesowej. Można powiedzieć, że DDD to po prostu dobrze zrobione programowanie obiektowe [Charlton, 2009].

Głównymi cechami programowania obiektowego są enkapsulacja i abstrakcja. Obie te cechy są silnie wykorzystywane w podejściu DDD.

DDD wprowadziło pojęcie bogatej domeny (ang. rich domain), która skupia się na enkapsulacji danych oraz na interakcji z otoczeniem jedynie przez wymianę komunikatów. Komunikaty te nie mogą polegać na wyciąganiu zawartości obiektów i ich modyfikacji z zewnątrz, gdyż wtedy naruszałyby prawo Demeter, wprowadzone przez Iana Hollanda [1987]. Bogata domena to tak naprawdę realizacja głównego założenia programowania obiektowego, które twórca tego paradygmatu, Alan C. Kay, definiuje następująco: „obiekty komunikują się ze sobą przez wysyłanie i odbieranie wiadomości" [1993].

Odbieranie i wysyłanie wiadomości prowadzi do kolejnej ważnej zasady projektowania bogatej domeny — segregacji polecenie-zapytanie (CQS, ang. command-query segregation), którą wprowadził i nazwał Bertrand Meyer w swojej pracy nad językiem obiektowym Eiffel [2012]. Głównym założeniem jest to, że „zadawanie pytania nie powinno zmieniać odpowiedzi". Polecenie wykonuje coś, ale nie zwraca rezultatu, a zapytanie zwraca rezultat, ale nie modyfikuje stanu.

Właściwe operowanie na abstrakcjach, a co za tym idzie również polimorfizmie i dziedziczeniu, to również domena DDD. Odpowiednio oddzielone koncepcje w różnych kontekstach związanych, projektowanie agregatów, czy używanie repozytoriów lub fabryk — to wszystko wymaga wydzielenia właściwych abstrakcji i korzystania z różnych ich poziomów. Ponadto, DDD w każdym miejscu, gdzie należy zastosować dobre praktyki programowania obiektowego, stara się stosować do zasad SOLID, które zostały zaproponowane, spopularyzowane i opisane przez Roberta C. Martina [2003].

Rozdział 3

Encje

Najważniejszą cechą encji jest posiadanie stałej, niezmiennej tożsamości przez cały cykl życia systemu. Encje muszą być rozróżnialne i unikalne. Dodatkowo, encje charakteryzuje zdolność do zmiany swojego stanu wewnętrznego. Możemy je utożsamiać z klasami posiadającymi stałą istotę, pomimo zmiennego stanu. Przykładami obiektów biznesowych tego typu będą:

- użytkownik — jego istota jest niezmienna, ale jego nazwa, e-mail, hasło, itp. mogą się zmieniać.
- produkt fizyczny — np. konkretny egzemplarz książki w księgarni lub magazynie. Produkt-książka w sklepie internetowym nie musi być zamodelowany jako encja — nie musi posiadać jednoznacznego identyfikatora, bo ten sam tytuł i wydanie na poziomie abstrakcji zakupu internetowego może określać tą samą książkę. Jednak, gdy dochodzi do sprzedania tej książki klientowi, to należy przynieść z magazynu jej konkretny egzemplarz. Ten konkretny egzemplarz może zmienić stan: status (np. na sprzedany), lokalizację, cenę (w związku z zagniecioną oprawą); nie zmienia jednak swojej tożsamości — to dalej ten sam egzemplarz.
- samochód — wymiana silnika lub zmiana właściciela nie powodują, że dany samochód został zmieniony w inny samochód.

3.1 Unikalna tożsamość

W celu zapewnienia stałej, unikalnej tożsamości encji, należy nadać jej odpowiedni identyfikator w momencie jej tworzenia i utrzymywać go niezmiennie przez cały cykl życia aplikacji. „Atrybut identyfikujący musi mieć gwarancję niepowtarzalności w systemie niezależnie od tego, jak ten system

jest zdefiniowany — nawet gdy jest rozproszony lub gdy obiekty są archiwizowane." [Evans, 2004]. Atrybut identyfikujący można stworzyć na wiele sposobów, podobnie jak na wiele sposobów można go zainicjalizować w encji.

3.1.1 Rodzaje identyfikatorów

Identyfikator naturalny (biznesowy)

Unikalnym identyfikatorem może być ustalony konkretny atrybut obiektu (np. numer PESEL) lub kombinacja jego atrybutów (numer wydania i tytuł dla gazety) [Evans, 2004]. Atrybuty wzięte pod uwagę jako identyfikujące muszą być niezmienne. Taki sposób ustalania identyfikatora ma jedną, główną zaletę — nie potrzeba tworzyć oraz korzystać z żadnego innego sztucznego pola w danej klasie. Istnieją jednak znaczne ograniczenia tego rozwiązania:

- nie można stworzyć obiektu bez tych danych (np. w celu ich późniejszego uzupełnienia).
- należy posiadać absolutną pewność, że atrybut identyfikujący jest unikalny i niezmienny. Przykładowy numer PESEL nie jest wcale unikalny [Stec-Fus, 2013], a także można go zmieniać zgodnie z prawem [Dz.U. 2010 nr 217, poz. 1427].
- nie można zmienić raz ustawionego atrybutu. Jeżeli np. użyjemy jako identyfikatora adres e-mail użytkownika, to nie będzie mógł go już nigdy zmienić. Ten sam użytkownik (fizycznie) zakładając konto z innym adresem e-mail będzie traktowany jako inny użytkownik.

Taki rodzaj identyfikatora należy stosować z wielką rozwagą. Przede wszystkim, należy świadomie spełnić powyższe wymagania, co wymaga dobrego zrozumienia logiki biznesowej. Trzeba wziąć pod uwagę, że raz podjętej decyzji dotyczącej wyboru atrybutu lub atrybutów nie będzie można łatwo, albo wcale zmienić.

Identyfikator sztuczny

Identyfikator sztuczny (klucz sztuczny, ang. surrogate key) to techniczny klucz główny danej encji. Charakteryzuje się tym, że nie powstaje z żadnego atrybutu obiektu, ale jest sztucznie do tego obiektu dodawany. Mimo, że nie pochodzi on z domeny biznesowej i przez to nie musi grać żadnej biznesowej roli, może zostać w takiej roli wykorzystany, np. ID przesyłki kurierskiej jest udostępniany użytkownikowi w celu śledzenia danej przesyłki [Evans, 2004]. Identyfikator sztuczny jest reprezentowany w aplikacjach głównie jako liczba lub UUID (ang. universally unique identifier).

Liczbowy klucz sztuczny jest generowany z zachowaniem kolejności oraz jest mniejszy co do wielkości bajtów, niż UUID (wielkość `bigint` w bazie PostgreSQL wynosi 8 bajtów [PostgreSQL, 2017], a wielkość UUID to 16 bajtów [Leach et al., 2005]). Zachowanie kolejności ma szczególne znaczenie. Z jednej strony, może nieść to ze sobą informację cenną dla użytkownika — kolejność utworzonych wpisów jest przechowywana w prosty sposób. Z drugiej strony, co bardziej istotne, wpływa korzystnie na fragmentacje indeksów klastrowanych, co może mieć istotny wpływ na wydajność odczytu i zapisu [Ford, 2014].

UUID to **uniwersalnie** unikatowy identyfikator, co sprawia, że jest on (szczególnie w wersji 4) niepowtarzalny nawet pomiędzy różnymi systemami. Jest to olbrzymia zaleta zwłaszcza w środowisku rozproszonym. W prosty sposób pozwala także na stworzenie i używanie identyfikatora w dowolnym miejscu systemu (front end, back end, ...), gdyż nie potrzeba żadnego sprawdzania i utrzymywania unikalności. Zmniejsza to złożoność przypadkową (ang. accidental complexity) systemu.

Używanie identyfikatora sztucznego jest praktyczną regułą. Zapewnia on unikalną tożsamość przez cały cykl życia encji niezależnie od zmian jej atrybutów. Sugeruję wybór UUID jako jego reprezentację. Obniżanie złożoności i możliwość tworzenia identyfikatora niezależnie od używanej infrastruktury to cechy, które sprawiają, że UUID jest najlepszym kandydatem na typ identyfikatora. Wydajność tego rozwiązania może być niedużo lub nawet niezauważalnie gorsza [Nilsson, 2002]. Warto wspomnieć, że np. w bazie NoSQL — MongoDB nie występują indeksy klastrowane, wobec czego nie wystąpi ta różnica w wydajności. Natomiast jeżeli faktycznie wydajność aplikacji stałaby się problemem, to można by się zastanowić nad rozwiązaniem hybrydowym, tzn. generować numeryczne ID w bazie danych, ale do utrzymywania unikalnej tożsamości encji używać osobnego pola z UUID.

3.1.2 Sposoby inicjalizacji

Klient zapewnia tożsamość

Identyfikator może być podawany z zewnątrz, przez klienta. Klienta, tzn. kod kliencki. Efektywnie może to być użytkownik, baza danych, narzędzie, inna aplikacja, itp. Z każdym rodzajem klienta wiąże się inna charakterystyka.

Jeżeli to użytkownik manualnie wprowadza identyfikator, to do aplikacji należy zawsze sprawdzenie jego unikalności. Takie sprawdzenie jednakże nie wyklucza możliwości popełnienia literówki przez użytkownika. Jeżeli identyfikatorem jest jakaś nazwa, to użytkownik może popełnić błąd w jej pisowni.

W związku z tym, że dany atrybut identyfikujący musi być niezmienny — błędu tego nie będzie można poprawić lub jego poprawa będzie kosztowna. W związku z tym, ten sposób powinien być unikany. Dane dostarczane przez użytkownika zawsze powinny być w jakiś sposób możliwe do modyfikacji, mimo iż mogą się one wydawać niezmienne [Vernon, 2013].

Sposób zapewniania tożsamości przez obcą aplikację (np. inny kontekst związany) ma dużo wspólnego z użytkownikiem wprowadzającym identyfikator. W takim rozwiązaniu, identyfikatory lokalnych encji są kopiowane z już istniejących zewnętrznych encji klienta. Przypomina to bardziej *mapowanie kontekstów* — jedno z pojęć DDD, które polega na tłumaczeniu znaczenia obiektów domenowych z jednego kontekstu, na podobne, wykorzystywane w innym kontekście. Strategia ta jest złożona i niestandardowa, wobec czego Vernon [2013] sugeruje, żeby używać tego podejścia tak zachowawczo jak to tylko możliwe.

Nadawanie identyfikatora przez bazę danych ma swoje uzasadnienie. Sekwencje bazodanowe są obiektami stworzonymi do zwracania unikalnej wartości. Wadą tego rozwiązania jest niewątpliwie wydajność. Komunikacja z bazą danych jest procesem kosztownym. Integracja z bazą danych w celu wygenerowania identyfikatora dokłada pewną złożoność do procesu tworzenia obiektu. Zdecydowanie utrudnia to testowanie obiektu domenowego.

Nadawanie takiego identyfikatora w kodzie polega na jego bezpośrednim przekazaniu w konstruktorze obiektu:

Listing 3.1: Encja z ID zapewnianym przez klienta

```
class User {
  private final UserId userId;
  private String firstName;
  private String lastName;

  /* UserId przekazywany w konstruktorze, co oznacza, że
  identyfikator jest dostarczany tej klasie przez klienta. */
  User(UserId userId, String firstName, String lastName) {
    this.userId = Objects.requireNonNull(userId);
    this.firstName = firstName;
    this.lastName = lastName;
  }
}
```

W tym miejscu należy wspomnieć o szczególnym przypadku nadawania identyfikatora przez bazę danych — przy zapisie encji w bazie danych. Te podejście, w oczywisty sposób, łamie zasadę nadawania identyfikatora encji w momencie jej tworzenia. Jest to jednak sposób na zlikwidowanie wad podejścia z nadawaniem identyfikatora przez bazę danych przy tworzeniu

obiektu. W tym przypadku nie używa się bazy danych przy każdym utworzeniu obiektu — co zmniejsza liczbę jej wywołań, oraz integracji z nią (np. w teście). Ponadto, dalej korzysta się z głównej zalety bazy danych — sekwencji, gdyż stworzona encja i tak zostanie ostatecznie zapisana w bazie danych, gdzie podczas tego procesu zostanie nadany jej identyfikator korzystając właśnie z sekwencji.

Poważnym minusem tego rozwiązania jest złamanie wspomnianej wyżej zasady, która zapewnia identyfikator przez cały cykl życia obiektu. Konsekwencje, które niesie to rozwiązanie, polegają na tym, że nie da się bezpiecznie porównywać obiektów danej klasy biorąc pod uwagę jedynie jej identyfikator. Ponadto, ten sam obiekt stworzony i niezapisany — nie jest tym samym obiektem po zapisaniu, gdyż wtedy dostaje dodatkowo ID. Takich obiektów nie da się użyć w kolekcjach wymagających niezmiennego kodu mieszającego (ang. hash code).

Komplikacje związane ze złamaniem zasady nadawania identyfikatora encji w momencie jej tworzenia są koncepcyjnie trudne do zapamiętania i płynnego używania. Mimo że te podejście jest ciągle często wykorzystywane w aplikacjach, w związku ze wskazanymi problemami, które niesie, rekomenduję odejście od tego sposobu przydzielania identyfikatora. Zauważam pozytywy wydajnościowe, które za nim stoją, ale w stosie zapisu (ang. Write) w większości przypadków, wydajność nie jest kluczowa, a kluczowa jest prosta i spójna koncepcja [Fowler, 2011].

Autogeneracja przez aplikację

Uzyskiwanie identyfikatora bezpośrednio przez jego autogenerację przez aplikację jest sposobem niezawodnym i bardzo wygodnym. Rodzaj identyfikatora w ten sposób uzyskany to najczęściej UUID lub jego odmiana. Główną zaletą tego rozwiązania jest samowystarczalność aplikacji w tworzeniu i utrzymywaniu niepowtarzalności identyfikatorów. To aplikacja steruje nadawaniem identyfikatorów i w związku z tym, w łatwy sposób może zapewniać encji jej tożsamość utrzymywaną od początku jej stworzenia.

Do tworzenia encji nie potrzeba żadnych zewnętrznych mechanizmów. W konstruktorze encji nie ma parametru ID, jest to szczegół implementacyjny, który jest poza kontrolą klienta:

Listing 3.2: Encja z autogenerowanym ID

```
class User {
  /* Można zainicjalizować ID przy polu, ale spójniej jest
    to zrobić razem z innymi polami - w konstruktorze. */
  private final UserId userId;
  private String firstName;
  private String lastName;

  /* UserId tworzony w konstruktorze, bez przepisywania go
    z parametru. */
  User(String firstName, String lastName) {
    userId = new UserId();
    this.firstName = firstName;
    this.lastName = lastName;
  }

  // ...
}
```

3.1.3 Opakowanie identyfikatorów

Każdy identyfikator, będący typem prostym lub generycznym, powinien zostać opakowany w obiekt wartości (patrz: rozdział 4) specyficzny dla encji, w której jest użyty. Dzięki temu możliwe jest statyczne typowanie identyfikatorów, które nie tylko broni programistę od pomyłki, ale też dostarcza mu dodatkowe informacje o używanym identyfikatorze. Jest to szczególnie istotne, gdy posługujemy się identyfikatorami jako referencjami danej encji w osobnych agregatach, co zostało omówione w rozdziale 5 („Agregaty"). Poniżej znajdują się przykłady identyfikatorów nieopakowanych i opakowanych:

```
// nieopakowany, generyczny identyfikator
Long id = 12L;
// opakowany identyfikator produktu
ProductId id = new ProductId(12L);

// analogicznie dla łańcuchów znakowych (UUID)
String id = "19d5cb02-d6ad-4193-a16b-3e6159bd9380";
UserId id = new UserId("19d5cb02-d6ad-4193-a16b-3e6159bd9380");
```

3.1.4 Implementacja

Unikalną tożsamość encji można uzyskać na wiele różnych sposobów. Jest to kluczowe zagadnienie, gdyż tożsamość tej samej encji nie może zostać ni-

gdy zmieniona. W świetle DDD, gdzie to model biznesowy powinien być realizowany jak najlepiej w klasach Java mu odpowiadających, najwłaściwszym sposobem tworzenia identyfikatorów jest ich autogeneracja przez aplikację i wykorzystanie do tego UUID. Ograniczenia lub wymagania techniczne mogą wymusić inne podejście tworzenia tożsamości encji, ale nie powinny być one traktowane jako wzór. Ponadto, w celu utrzymania bogatej domeny, każdy identyfikator powinien być opakowany w odpowiadający mu obiekt wartości charakterystyczny dla danej encji.

Poniżej znajduje się przykładowa implementacja identyfikatora (obiektu wartości, zob. rozdział 4) `UserId` oraz encji `User`. Na szczególną uwagę zasługują metody `equals()` oraz `hashCode()`, które zapewniają unikalną tożsamość zarówno klasie identyfikatora, jak i danej encji.

Listing 3.3: Klasa identyfikatora `UserId`

```
import java.util.UUID;

public class UserId {
  private final String value;

  // konstruktor dla nowo tworzonego obiektu
  public UserId() {
    value = UUID.randomUUID().toString();
  }
  // konstruktor służący rekonstytucji obiektu
  public UserId(String id) {
    value = UUID.fromString(id).toString();
  }

  @Override
  public boolean equals(Object o) {
    if (this == o) {
      return true;
    }
    if (!(o instanceof UserId)) {
      return false;
    }
    UserId userId = (UserId) o;
    return value.equals(userId.value);
  }

  @Override
  public int hashCode() {
    return value.hashCode();
  }
}
```

Listing 3.4: Encja `User`

```
class User {
  private final UserId userId;
  private String firstName;
  private String lastName;

  User(String firstName, String lastName) {
    userId = new UserId();
    this.firstName = firstName;
    this.lastName = lastName;
  }

  @Override
  public boolean equals(Object o) {
    if (this == o) {
      return true;
    }
    if (!(o instanceof User)) {
      return false;
    }
    User user = (User) o;
    return userId.equals(user.userId);
  }

  @Override
  public int hashCode() {
    return userId.hashCode();
  }

  // ...
}
```

3.2 Interakcje

Używanie encji polega na jej modyfikowaniu z zewnątrz, z wewnątrz oraz na zwracaniu informacji przez nią przechowywanych. Dzięki zachowaniu jednoznacznej tożsamości, encja może całkowicie zmieniać swój stan nadal pozostając tą samą encją.

Mutowalność (ang. mutability — zdolność do zmiany) to istotna cecha encji. Ten sam obiekt musi mieć możliwość zmiany swojego stanu. Te zmiany są ograniczone jedynie regułami biznesowymi, zwanymi niezmiennikami. Wg. DDD obiekty biznesowe, takie jak encje, muszą być zawsze spójne — od momentu stworzenia, przez cały okres życia. Z tego powodu, każdą zmianę stanu obiektu należy walidować zgodnie z niezmiennikami tego obiektu.

W kolejnych sekcjach zostały opisane reguły, którymi należy się kierować, żeby projektować encje zgodnie z założeniami wprowadzonymi przez pojęcia bogatej domeny oraz zasadą CQS opisanymi w rozdziale 2 („Domain-Driven Design").

3.2.1 Implementacja poleceń

Mutowalność

Przy stosowaniu popularnej anemicznej domeny, gdzie klasy są praktycznie strukturami danych, a nie obiektami, za zmianę stanu klasy odpowiadają wyłącznie settery, czyli metody `setX(x)`, gdzie X jest atrybutem klasy, a x jest nową wartością na jaką należy ustawić ten atrybut. W DDD jakakolwiek zmiana stanu powinna wychodzić od konkretnego polecenia (ang. command; C w CQS) mającego biznesowe uzasadnienie.

W związku z powyższym, settery nie powinny być nigdy publiczne, nawet jeżeli wykonują więcej niż prostą logikę przypisującą nową wartość. Po publicznych metodach z prefiksem *set-* klient klasy nie spodziewa się niczego więcej niż prostego przypisania wartości, wobec czego zmiana tego zachowania spowodowałaby złamanie zasady POLA (ang. principle of least astonishment, pol. zasada najmniejszego zaskoczenia; [Raymond, 2003]). Jeżeli faktycznie metoda implementuje jedynie proste przypisanie nowej wartości do atrybutu, to proponuję by zastąpić prefiks *set-*, prefiksem *change-*. Słowo to nie kojarzy się ze szkodliwym dla bogatej domeny standardem JavaBeans [Hamilton, 1997]. Prywatne settery w obrębie klasy są dopuszczalne i czasem pożądane. W konstruktorze klasy istnieje często potrzeba przypisania wartości danego atrybutu w ten sam sposób, w jaki zmienia się tą wartość przez metodę *change-*. Dziwnie by jednak wyglądało, gdyby podczas tworzenia obiektu danej klasy używać metody *change-* sugerującej zmianę obiektu, którego jeszcze nia ma. Ponadto, metoda *change-* może mieć dodatkową logikę związaną ze zmianą danego atrybutu — np. nie może pozwolić na zmianę danego atrybutu, jeśli nie zmienił się jakiś inny.

Walidacja

Walidacja obiektu domenowego ma na celu utrzymanie jego spójności z logiką biznesową przez cały cykl jego życia. Stosowanie tej zasady daje gwarancję spójności obiektu; kod kliencki nie musi wykonywać dodatkowych sprawdzeń („ifów"), co zmniejsza możliwości popełnienia błędu oraz ułatwia programowanie na różnych poziomach abstrakcji. Dzięki zastosowaniu niezmienników (reguł biznesowych) bezpośrednio w klasach modelujących do-

menę, gdy nastąpi potrzeba zmiana tej reguły, to modyfikacji ulegnie jedynie ta jedna klasa. Klient, używający tej klasy w wielu miejscach, nie będzie zmuszony do żadnej zmiany.

W przypadku utrzymywania spójności domeny biznesowej możemy mówić o czerech rodzajach walidacji (Vernon [2013] wyróżnia trzy pierwsze z nich):

- walidacja atrybutu obiektu biorąca pod uwagę jedynie ten atrybut;
- walidacja atrybutu obiektu biorąca pod uwagę również inne atrybuty;
- walidacja kolekcji obiektów;
- walidacja biorąca pod uwagę zewnętrzne czynniki.

Najprostsza walidacja dotyczy jedynie pojedynczego atrybutu. Najczęściej jest to sprawdzenie, czy dany atrybut nie jest równy `null` lub jeśli dany atrybut to kolekcja — czy nie jest ona pusta.

Walidacja atrybutu biorąca pod uwagę inne atrybuty oznacza, że na stan danego atrybutu wpływają inne atrybuty obiektu. W tak walidowanej metodzie znajduje się zapytanie o stan tego innego elementu, który pozwala lub nie na kontynuację wywołania metody. Stan pytanego elementu może się zmienić, jeżeli taka jest logika biznesowa. Nie może on jednak zmienić się w metodzie „pytającej", gdyż byłoby to pogwałcenie zasady CQS.

Walidacja kolekcji obiektów ma miejsce wtedy, gdy mimo że dany obiekt sam w sobie jest spójny (zwalidowany), to niepoprawne jest jego wystąpienie w istniejącej kolekcji tych obiektów. Przykładem może być tu unikatowość jakiegoś atrybutu obiektu w ramach kolekcji obiektów lub ustalony limit obiektów danej kolekcji — próba dodania kolejnego obiektu poza limit, mimo że spójnego i zwalidowanego, będzie skutkowała błędem walidacji takiej metody.

Z walidacją biorąca pod uwagę zewnętrzne czynniki mamy do czynienia, gdy niezależnie od stanu danego komponentu operacja nie może się powieść. Przykładem może być sytuacja uzależnienia wykonania metody biznesowej od czasu. Jeżeli modyfikacja danego obiektu lub jego atrybutu jest dozwolona jedynie, gdy jest on ważny w przyszłości, to czynnik zewnętrzny — czas teraźniejszy — będzie powodował powodzenie lub niepowodzenie walidacji.

Przykład

Poniżej znajduje się implementacja encji `Account`, w której zaprezentowano powyższe zasady w praktyce:

Listing 3.5: Encja `Account`

```
class Account {
```

```
private final AccountId id;
private Status status;
private Limit limit;
private Transfers transfers;

Account(Limit limit) {
  id = new AccountId();
  status = Status.INACTIVE; // domyślna wartość początkowa
  setLimit(limit); // użycie prywatnego settera w konstruktorze
  transfers = new Transfers();
}

private void setLimit(Limit limit) {
  // walidacja pojedynczego atrybutu
  Objects.requireNonNull(limit, "Limit mustn't be null");
  if (limit.isBelowDefault()) {
    throw new IllegalArgumentException(
        "Limit cannot be below default");
  }
  this.limit = limit;
}

// metoda biznesowa, która nie wymaga podawania argumentów
void activate() {
  status = Status.ACTIVE;
}

// metoda biznesowa ,,change-'' zmieniająca stan na podany
void changeLimit(Limit limit) {
  checkActivity(); // dodatkowe sprawdzenie przy zmianie
  setLimit(limit); // użycie tej samej metody co w konstruktorze
}

// metoda do sprawdzania innego atrybutu - aktywności
private void checkActivity() {
  if (status.equals(Status.INACTIVE)) {
    throw new IllegalStateException(
        "Can't change inactive account");
  }
}

void makeTransfer(Transfer transfer) {
  checkActivity();
  // walidacja kolekcji elementów
  if (limit.isExceededBy(transfers.balanceWith(transfer))) {
    throw new IllegalStateException("Can't add transfer");
  }
  transfers.add(transfer);
}
```

```
  void removeTransfer(TransferId transferId,
                      TimeProvider timeProvider) {
    checkActivity();
    Transfer transfer = transfers.get(transferId);
    // walidacja biorąca pod uwagę zewnętrzne czynniki
    if (transfer.isNotAfter(timeProvider.today())) {
      throw new TransferInThePast();
    }
    transfers.remove(transferId);
  }

  private enum Status { ACTIVE, INACTIVE }

  // ...
}
```

Konstruktor klasy `Account` przyjmuje jedynie jeden parametr, który klient tworzący tą klasę musi podać. Pozostałe atrybuty tej klasy (jej struktura wewnętrzna) są ukryte przed klientem, zgodnie z zasadami enkapsulacji. W konstruktorze klasy tworzona jest jego tożsamość oraz ustawiane są wartości początkowe. Oprócz tego, wywoływana jest metoda `setLimit(Limit)`, która jest prywatnym setterem wykorzystywanym zarówno w konstruktorze jak i w metodzie `changeLimit(Limit)`, która jest udostępniona na zewnątrz klasy — klientom w obrębie pakietu. Metoda ta, poza ustawieniem limitu, sprawdza jeszcze dodatkowo, czy inny atrybut encji jest w stanie pozwalającym na jego zmianę.

Metoda `makeTransfer(Transfer)` zawiera walidację kolekcji elementów. Mimo, iż pojedynczy obiekt `Transfer` jest prawidłowy, to od kolekcji takich obiektów wymaga się, żeby nie przekroczyły danego limitu — inaczej kolekcja byłaby nieprawidłowa.

Walidacja biorąca pod uwagę zewnętrzny czynnik — obecną datę — znajduje się w metodzie `removeTransfer(TransferId, TimeProvider)`. Warto zwrócić uwagę, że zależność od tego czynnika jest przekazana jako dodatkowy parametr metody i stanowi jej „domknięcie” (ang. closure). Można by zamiast tego podać tę datę bezpośrednio, ale wtedy odpowiedzialność za wybranie odpowiedniej daty zostanie przesunięta na klienta. Najczęściej nie jest to odpowiednie, gdyż to w obiekcie domenowym powinna być zawarta informacja, jaka data jest datą graniczną dla usunięcia przelewu. Przykładowo, gdy domyślna data graniczna zmieni się z „dziś” na „jutro” to zmiana będzie potrzebna jedynie w jednym miejscu — obiekcie domenowym. Gdyby informacja o dacie granicznej była podawana przez klienta to zmiana byłaby wymagana w każdym miejscu, gdzie klient wywołuje metodę `removeTransfer()`.

3.2.2 Implementacja zapytań

Metody realizujące zapytania należy tworzyć bardzo ostrożnie. Ich duża liczba może prowadzić do proceduralnego stylu programowania oraz anemicznej domeny [Fowler, 2003b]. Jak słusznie zauważa Vernon [2013], jeżeli da się po prostu dostęp do danych obiektów domenowych, to klasy te staną się strukturami danych obdartymi z zachowań.

Niestety, popularne *gettery* są jednym z najbardziej rozpowszechnionych anty-wzorców programowania obiektowego. Spopularyzowane przez specyfikację JavaBeans [Hamilton, 1997] metody o prefiksie *get-* są często bez przemyślenia generowane dla wszystkich pól klasy — co w praktyce łamie enkapsulację. Dodatkowo, prefiks *get-* nie dodaje prawie żadnej wartości znaczeniowej, a wprowadza dodatkowo niepotrzebny wyraz. Standard JavaBeans był wykorzystywany przez niektóre narzędzia, takie jak Hibernate, ale obecnie bardzo dobrze radzą sobie one z dostępem do pól poprzez refleksję, wobec czego można w końcu ostatecznie skończyć implementować ten szkodliwy standard, zwłaszcza tworząc aplikację zgodnie z podejściem DDD. Pierwszym krokiem w dobrą stronę jest chociażby porzucenie prefiksu *get-* w nazwie metody.

Projektowanie metod dostępowych polega na dobraniu odpowiedniej nazwy, wartości zwracanej oraz parametrów. Tak jak zostało to wspomniane wyżej, nazwa nie powinna się zaczynać od prefiksu *get-*, gdyż sugeruje to użycie specyfikacji JavaBeans. Wartość zwracana nie musi być tożsama z posiadanym przez klasę polem, ale jeżeli jest, to należy zagwarantować, że nie da się zmienić wartości tego pola przez tą metodę. Parametrów metody dostępowej powinno być jak najmniej. Powinno dążyć się do braku parametrów. Złym zwyczajem jest przekazywanie tzw. flag — zamiast tego powinno się tworzyć inaczej nazwane oddzielne metody.

Poniżej przykład jak nie powinno się implementować zapytań do encji:

Listing 3.6: Przykład nieprawidłowej implementacji zapytań

```
class User {
  private final UserId userId;
  private String firstName;
  private String lastName;
  private final Roles roles;

  User(String firstName, String lastName) {
    userId = new UserId();
    this.firstName = firstName;
    this.lastName = lastName;
    roles = new Roles();
  }
```

```
  String getName(boolean first) {
    if (first) {
      return firstName;
    }
    return lastName;
  }

  Roles getRoles() {
    return roles;
  }

  // ...
}

class UserClientCode {
  void queryTheUser() {
    User user = new User("Janusz", "Testowski");

    String firstName = user.getName(true);
    String lastName = user.getName(false);
    String fullName = firstName + " " + lastName;

    Roles roles = user.getRoles();
    Set<String> roleNames = roles.names();
    roles.remove(Role.USER);
  }
}
```

Przykład ten ilustruje błędnie zaprojektowane metody dostępowe, którym dobrano nieodpowiednie:

- nazwy. Wszystkie zaczynają się od prefiksu *get-*, który nic nie wnosi.
- parametry. Wyjście metody `getName(boolean first)` zależy od flagi, która wprowadza niepotrzebną złożoność (ang. accidental complexity).
- wartość zwracaną.

Źle dobrana wartość zwracana jest najgorszym błędem. Wynika ona z braku znajomości przypadku użycia dla którego wynikła potrzeba dodania metody pytającej lub nieświadomego wystawienia kodu klasy `User` na niebezpieczeństwo nieodpowiedniej zmiany zawartości tej klasy przez kod kliencki. Skutkiem takiego błędu jest przeniesienie odpowiedzialności na kod kliencki danej klasy, co może spowodować utratę spójności encji i nieprawidłowe działanie programu. Nawet jeżeli kod kliencki zadba o spójność, to przesunięcie logiki z obiektu domenowego, na kod kliencki powoduje, że staje się on proceduralny i trudny w utrzymaniu. Każda zmiana logiki dotycząca tych metod

klasy `User` spowoduje, że wszędzie, gdzie są one używane — kod kliencki będzie musiał się zmienić.

W powyższym przykładzie jednym z przypadków użycia jest otrzymanie nazwy użytkownika — jego imienia i nazwiska. Łączenie imienia i nazwiska nie powinno być odpowiedzialnością kodu klienckiego, gdyż jest to efekt końcowy, który mógłby otrzymać wołając odpowiednią metodę klasy `User`, która operując na swoich danych potrafiłaby zwrócić odpowiednią odpowiedź.

Innym przypadkiem użycia jest otrzymanie wszystkich nazw ról użytkownika. Tu znowu brak odpowiedniej metody realizującej właśnie to wymaganie. Dodatkowo, należy zwrócić uwagę, że poprzez zwrócenie referencji do obiektu `Roles` klient jest w stanie dokonać akcji, która może rozszczelnić encję `User` (złamać jej niezmienniki). Być może usunięcie roli `Role.USER` z encji `User` nie jest dozwolone lub powoduje, że użytkownik ten zmienia swój status. Poprzez wyciągnięcie zawartości encji w dany sposób, encja nie jest w stanie zapewnić swoją spójność.

Poniżej przykład jak właściwie powinny zostać zrealizowane dane metody dostępowe:

Listing 3.7: Przykład prawidłowej implementacji zapytań

```
class User {

  // ...

  String name() {
    return firstName + " " + lastName;
  }

  Set<String> roleNames() {
    return roles.names();
  }

  // ...
}

class UserClientCode {
  void queryTheUser() {
    User user = new User("Janusz", "Testowski");

    String username = user.name();

    Set<String> roleNames = user.roleNames();
  }
}
```

Logika biznesowa dot. sklejania nazwy użytkownika została ukryta w encji. Podobnie jak role, które zostały skutecznie zenkapsulowane — kod kliencki nie musi nawet wiedzieć jak zaimplementowane są role w encji `User`, żeby zdobyć ich nazwy. Liczba parametrów została zminimalizowana, a nazwy metod są bardziej zwięzłe i konkretne.

Zwracanie DTO

W języku Java metoda może zwrócić tylko jeden obiekt. Problem pojawia się więc, gdy zachodzi potrzeba zwrócenia większej liczby obiektów pojedynczym zapytaniem. Powinno się unikać zwracania obiektów domenowych kodu klienckiemu spoza danego kontekstu (pakietu), np. w celu zaprezentowania danych przechowywanych w tym obiekcie domenowym. Do tego celu świetnie nadają się struktury danych.

Struktury danych w języku Java są również klasami, tak jak obiekty domenowe, ale w odróżnieniu od nich mogą być domyślnie publiczne, posiadać gettery i settery (a nawet pola publiczne). Należy je jednak traktować jedynie jako pojemnik na dane nie posiadający zachowań. Tak rozumiana struktura danych jest często nazywana DTO.

DTO (ang. Data Transfer Object, pol. obiekt służący do transferu danych) to termin ukuty przez Martina Fowlera [2003a]. Początkowo, odnosił się jedynie do scalania danych z kilku obiektów celem zmniejszenia liczby wymaganych zdalnych wywołań zewnętrznych usług, ale obecnie mówi się o nim bardziej w kontekście struktury danych podróżującej (ang. transfer) przez różne warstwy aplikacji.

Jednym ze sposobów realizacji zapytań jest właśnie zwracanie DTO z każdej encji. Dzięki temu można pozbyć się innych metod dostępowych, gdyż DTO jest konstruowane przez daną klasę zawierającą dane pozwalające na jego stworzenie, co powoduje zwiększenie enkapsulacji. Takie rozwiązanie, jak sugeruje Sławomir Sobótka [2008] może także „stanowić warstwę interfejsu do systemu pod którą możemy swobodnie zmieniać nasz model bez obawy o ciągłe modyfikacje klientów.". Sobótka w swoim projekcie referencyjnym [2013] stworzył w niektórych klasach domenowych metody `generateSnapshot()`, które generują struktury danych na podstawie informacji zawartych w klasach domenowych. Te struktury danych są po prostu klasami z sufiksem *-Data*, np. `ClientData`, `ProductData`.

Te podejście jest też polecane przez Jakuba Nabrdalika, który w swojej prezentacji [2017] wykorzystuje DTO w roli interfejsu danego pakietu, z którego można korzystać jedynie poprzez publiczną klasę — *fasadę* [Gamma et al., 1994]. Dzięki zastosowaniu takiego rozwiązania uzyskuje się odpowiednią enkapsulację również na poziomie pakietów, sprawiając, że większość klas

staje się prywatna w obrębie pakietu, co jest, nie bez powodu, domyślną wartością modyfikatora dostępu w języku Java.

Przykład implementacji tego podejścia znajduje się poniżej.

Listing 3.8: Przykład implementacji podejścia ze zwracaniem obiektów DTO z encji

```
class User {
  private final UserId userId;
  private String firstName;
  private String lastName;
  private Roles roles;

  // konstruktor i inne metody

  UserDto toDto() {
    return new UserDto(
        firstName + " " + lastName,
        roles.toDto()
    );
  }

  // ...
}

public class UserDto {

  private String name;
  private RolesDto rolesDto;

  public UserDto(String name, RolesDto rolesDto) {
    this.name = name;
    this.rolesDto = rolesDto;
  }

  // gettery, settery, equals i hashCode
}

public class UserFacade {

  // ...

  public UserDto createUser() {
    User user = new User("Janusz", "Testowski");
    return user.toDto();
  }
}
```

```
class UserClientCode {

  // ...

  void queryTheUser() {
    UserDto userDto = userFacade.createUser();
    String username = userDto.getName();
    Set<String> roleNames = userDto.getRolesDto().getNames();
  }
}
```

Warto zwrócić uwagę, że klasa `UserDto` nie jest odzwierciedleniem wszystkich danych klasy `User`, gdyż nie jest to potrzebne, a mogłoby być ograniczające co do implementacji klasy `User`. `UserDto` jest publiczną strukturą danych, a nie klasą domenową, wobec czego gettery i settery nie są tam problemem. Klasa `UserDto` nie ma i nie powinna mieć żadnych metod biznesowych, a jedynie ewentualnie może posiadać proste walidacje (np. czy dana wartość nie jest równa `null`).

Przy tworzeniu instancji `UserDto` ważne jest, żeby klasa `User` nie użyła żadnej modyfikowalnej wartości do jej stworzenia i tak się dzieje. Złączenie obiektów klasy `String` jest niemutowalnym obiektem klasy `String`, a metoda `Roles.toDto()` podlega tym samym zasadom tworzenia DTO i nie używa żadnych referencji do swoich wnętrzności. Warto zauważyć, że metody `toDto()` muszą istnieć dla każdej skomponowanej klasy, gdyż klasa komponująca może mieć potrzebę ją zawołać, żeby sama móc zbudować swoje DTO.

Linijka kodu w klasie `UserClientCode`, która wyciąga nazwy ról poprzez najpierw wyciąganie `RolesDto`, a potem na tym DTO wywołuje metodę `getNames()` to przykład, gdzie można by wprowadzić pewną metodę dodatkową do klasy DTO. Można by te dwa zapytania ukryć w jedno. Jednakże, nie jest to niezbędne, gdyż klasa ta nie jest obiektem domenowym, więc nie podlega zasadom programowania obiektowego per se.

Rozdział 4

Obiekty wartości

Obiekt wartości (ang. Value Object) to wg. Evansa [2004] „obiekt reprezentujący opisowy aspekt dziedziny bez żadnej tożsamości z nim związanej (...). Obiekty te tworzone są jedynie w celu reprezentacji elementów projektu, które mają dla nas znaczenie tylko z tego powodu, jakie są, a nie kim lub czym są.". Wg. Vernona [2013] obiekt wartości cechuje to, że:

a. mierzy, określa lub opisuje coś w domenie;
b. modeluje koncepcyjną całość, komponując powiązane atrybuty jako integralną jednostkę;
c. może być utrzymywany jako niezmiennik (obiekt niemutowalny);
d. może być całkowicie wymieniony, gdy jego pomiar lub opis zmienia się;
e. można go porównać z innymi za pomocą równości wartości;
f. dostarcza swoim współpracownikom zachowania bez skutków ubocznych.

Powyższe cechy można podzielić na dwie grupy tematyczne. Punkty *a* i *b* dotyczą celu wyodrębnienia obiektu wartości jako osobnego bytu domenowego. Cechy *c*–*f* traktują o tym, jak obiekt wartości powinien być implementowany i używany.

4.1 Cel

Obiekt wartości modeluje byt biznesowy, który nie wymaga tożsamości. Brak tożsamości redukuje złożoność koncepcyjną modelu, co jest pożądane. Obiekty wartości powinny być domyślnymi bytami domenowymi. Ich przykładami w bibliotece standardowej są klasy `String` i `BigDecimal`.

Domenowy obiekt wartości opakowuje techniczne składowe udostępniając metody będące częścią języka wszechobecnego (ang. ubiquitous language). Dla klientów obiektu wartości nieistotne są jego szczegóły implementacyjne, a jedynie to, co on może zrobić. Klient klasy `String` nie ma dostępu do wewnętrznej tablicy `char[]` i nie może na niej wykonywać żadnych operacji — ta tablica jest opakowana przez abstrakcję klasy `String`.

Taką samą zasadą należy się posłużyć projektując własne klasy domenowe. W miarę możliwości należy ograniczać interfejs klasy domenowej i precyzyjnie określać jej odpowiedzialności.

Przykładem odpowiedniego obiektu wartości jest np. klasa `Limit`, użyta w jednym z poprzednich przykładów w rozdziale 3 („Encje"), użyta w encji `Account`:

Listing 4.1: Klasa `Limit` użyta w encji `Account`

```
class Account {
  private final AccountId id;
  private Status status;
  private Limit limit;
  private Transfers transfers;

  //...
}

class Limit {

  private final BigDecimal value;

  //...
}
```

Klasa `Limit` przechowuje jedynie jedno pole typu `BigDecimal`. Istnieje pokusa, żeby wobec tego zamodelować limit w klasie `Account` jako składowa `BigDecimal limit`. Jednakże, takie podejście byłoby niezgodne z DDD, gdyż obiekt wartości `Limit` precyzyjnie określa pojęcie domenowe „limit" i to, co można z nim zrobić. Jeżeli klasa `Account` użyłaby zamiast konkretnego obiektu wartości, obiektu limit o typie `BigDecimal`, to mogłaby z nim zrobić wszystko to, co można zrobić z obiektem klasy `BigDecimal`. Wprowadziłoby to niepotrzebną złożoność, a także przesunęłoby odpowiedzialność na implementację logiki związanej z limitem na ciało klasy `Account`.

Obecnie interfejs, czyli dostępne zachowania, klasy `Limit` pozwala jedynie na utworzenie jej obiektu, sprawdzenie, czy jest on poniżej wartości domyślnej (`isBelowLimit()`) oraz czy limit jest przekroczony przez podaną wartość (`isExceededBy(BigDecimal value)`):

Listing 4.2: Implementacja obiektu wartości `Limit`

```
class Limit {
  private final BigDecimal value;

  private final static BigDecimal DEFAULT_VALUE
      = new BigDecimal(1_000);

  Limit(BigDecimal value) {
    this.value = value;
  }

  boolean isBelowDefault() {
    return value.compareTo(DEFAULT_VALUE) < 0;
  }

  boolean isExceededBy(BigDecimal value) {
    return value.compareTo(actualValue()) < 0;
  }

  private BigDecimal actualValue() {
    return value.negate();
  }
}
```

Klient nie może na takim obiekcie wykonać operacji, które mógłby na obiekcie klasy `BigDecimal`, np. dodać czy pomnożyć. Jest to zasadne, gdyż w domenie biznesowej nie występuje pojęcie mnożenia limitów, wobec czego udostępnienie takiej operacji klientowi mogłoby prowadzić do błędów lub trudności w zrozumieniu domeny. Warto zwrócić również uwagę na fakt, że do określania, czy limit jest przekroczony wykorzystuje się ujemną wartość limitu. Jest to wiedza biznesowa — limit 1000 zł oznacza, że wartości, które są większe lub równe -1000 zł nie przekraczają tego limitu. Jeżeli taką wiedzę przenieślibyśmy do encji `Account` to stanowiłaby ona zdecydowanie dziwny koncept w tamtym miejscu, o którym zawsze trzeba by było pamiętać. W obiekcie wartości `Limit` ma ona jednak całkowity sens i klient tego obiektu nie potrzebuje wiedzieć dokładnie jak on działa.

4.1.1 Klasa opakowująca kolekcje obiektów

Typem, który często nie jest, a powinien być modelowany jako obiekt wartości jest kolekcja obiektów. Kolekcje obiektów w języku Java posiadają wiele różnych metod, które nie są odpowiednie dla modelowanej domeny i niepożądanie przenoszą odpowiedzialność na klienta. Poniżej przykład klasy `Transfers`, która jest opakowaniem kolekcji obiektów klasy `Transfer` i przez

to — obiektem wartości:

Listing 4.3: Klasa `Transfers` — opakowanie kolekcji obiektów klasy `Transfer`

```
class Transfers {
  private final Set<Transfer> transfers = new HashSet<>();

  Transfer get(TransferId transferId) {
    return transfers
      .stream()
      .filter(transfer -> transfer.hasId(transferId))
      .findAny()
      .orElseThrow(() -> new TransferNotFound(transferId));
  }

  void add(Transfer transfer) {
      transfers.add(transfer);
  }

  void remove(TransferId transferId) {
      transfers.remove(get(transferId));
  }

  Balance balanceWith(Transfer transfer) {
      return new Balance(sum().add(transfer.amount()));
  }

  private BigDecimal sum() {
    return transfers
      .stream()
      .map(Transfer::amount)
      .reduce(BigDecimal.ZERO, BigDecimal::add);
  }
}
```

Tak opakowana kolekcja udostępnia zaledwie kilka operacji. W niektórych miejscach jej implementacja jest trywialna, tak jak opakowana metoda `add(Transfer)`, ale gdzie indziej wspiera operacje na kolekcjach, które jedynie zaciemniały by obraz klientowi zwykłej kolekcji, jak np. metoda `get(TransferId)`. Dodatkowo, oprócz typowych metod związanych z kolekcjami, wprowadzona została metoda `balanceWith(Transfer)`, która umieszcza logikę biznesową związaną z sumowaniem transakcji w odpowiednim miejscu — kontenerze transakcji. Taka klasa opakowująca kolekcję daje jeszcze jedną zaletę płynącą z odpowiedniej enkapsulacji — swobodną wymianę implementacji. Wymiana implementacji kolekcji w klasie `Transfers` na listę lub mapę będzie przezroczysta dla klientów tej klasy.

W tym miejscu warto wspomnieć o łatwym sposobie umożliwienia iterowania po klasie opakowującej kolekcje. W tym celu wystarczy zaimplementować interfejs `Iterable` poprzez implementację metod `iterator()` oraz `spliterator()` wykorzystując opakowywaną kolekcję jako delegat. Należy jednak pamiętać, że oddaje to w użycie klientowi potężne narzędzie, które może powinno być zaimplementowane jako metoda klasy opakowującej.

Listing 4.4: Przykład klasy opakowującej kolekcje z zaimplementowanym interfejsem `Iterable`

```
class Transfers implements Iterable<Transfer> {
  private final Set<Transfer> transfers = new HashSet<>();

  // ...

  @Override
  public Iterator<Transfer> iterator() {
    return transfers.iterator();
  }

  @Override
  public Spliterator<Transfer> spliterator() {
    return transfers.spliterator();
  }
}
```

4.1.2 Wiele klas podobnego typu

Obiekty wartości stanowią dobre opakowanie dla tzw. „wartości prostej" w obiekt domenowy. Powinny być wykorzystywane jak najczęściej, nawet gdy opakowują zaledwie jedną wartość. Zmniejsza to możliwość popełnienia błędu przez programistę, gdyż zwiększa to liczbę typów, które w Javie są sprawdzane na poziomie kompilacji (poprzez tzw. statyczne typowanie). Takie podejście stanowi jednak niebezpieczeństwo użycia zbyt dużej liczby klas dla podobnych typów. Nazwa użytkownika różni się od nazwy firmy miejscem zastosowania, ale oba te obiekty to nazwy.

Wobec tego problemu proponuję użycie **tego samego obiektu wartości**, jeżeli jego zachowania są **takie same**, a **różnych obiektów** — gdy **zachowania się różnią**.

Przykładem obiektów wartości, które mają te same zachowania jest wartość monetarna (ang. `MoneyValue`). Taka klasa powinna pozwolić np. na dodawanie pieniędzy, ale tylko takiej samej waluty. Można więc zamodelować ten problem wykorzystując dwa różne obiekty wartości:

Listing 4.5: Przykład klas podobnego typu jako dwa różne obiekty wartości

```
class UsdMoneyValue {
  private final BigDecimal value;
  private final Currency currency = Currency
      .getInstance("USD");

  UsdMoneyValue(BigDecimal value) {
    this.value = value;
  }

  public UsdMoneyValue add(UsdMoneyValue other) {
    return new UsdMoneyValue(value.add(other.value));
  }

  // ...
}

class EurMoneyValue {
  private final BigDecimal value;
  private final Currency currency = Currency
      .getInstance("EUR");

  EurMoneyValue(BigDecimal value) {
    this.value = value;
  }

  public EurMoneyValue add(EurMoneyValue other) {
    return new EurMoneyValue(value.add(other.value));
  }

  // ...
}
```

Oba obiekty wartości udostępniają takie samo zachowanie — dodawanie wartości pieniężnej. Dzięki temu, że są osobnymi typami, to niemożliwa (nie dająca się skompilować) jest sytuacja, gdzie błędnie dodamy dolary do euro. Takie podejście generuje jednak wiele klas mających te same zachowania. Ten problem można by rozwiązać jedną klasą:

Listing 4.6: Przykład tej samej klasy obiektów wartości

```
class MoneyValue {
  private final BigDecimal value;
  private final Currency currency;

  MoneyValue(BigDecimal value, Currency currency) {
    this.value = value;
    this.currency = currency;
  }

  public MoneyValue add(MoneyValue other) {
    checkIfCurrencyTheSame(other);
    return new MoneyValue(value.add(other.value), currency);
  }

  private void checkIfCurrencyTheSame(MoneyValue other) {
    if (!currency.equals(other.currency)) {
      throw new DifferentCurrencyException(
          currency, other.currency);
    }
  }

  // ...
}
```

Traci się w ten sposób statyczne rozróżnienie, jaka wartość pieniężna dotyczy jakiej waluty, ale istnieje to samo zachowanie każdej z wartości pieniężnej, co pozwala na zamodelowanie tego jedną klasą obiektów wartości.

Z inną sytuacją ma się do czynienia, gdy istnieją różne zachowania klas o podobnym typie. Taka sytuacja może wystąpić modelując np. zakres dat (ang. `DateRange`). Mimo to, że dla każdego zakresu dat można określić datę początkową i końcową, to inaczej można traktować zakres dat rozciągający się na wiele miesięcy, a inaczej zakres dat będący pojedynczym miesiącem. Oto przykłady:

Listing 4.7: Przykład obiektu wartości modelującego zakres dat rozciągający się na wiele miesięcy

```
class AnyDateRange implements DateRange {
  private final LocalDate startDate;
  private final LocalDate endDate;

  AnyDateRange(LocalDate startDate, LocalDate endDate) {
    Objects.requireNonNull(startDate);
    Objects.requireNonNull(endDate);
    if (startDate.isAfter(endDate)) {
      throw new StartDateAfterEndDateException();
    }
    this.startDate = startDate;
    this.endDate = endDate;
  }

  @Override
  public LocalDate startDate() {
    return startDate;
  }

  @Override
  public LocalDate endDate() {
    return endDate;
  }

  // ...
}
```

Listing 4.8: Przykład obiektu wartości modelującego zakres dat będący pojedynczym miesiącem

```
class YearMonthDateRange implements DateRange {
  private final LocalDate startDate;
  private final LocalDate endDate;

  YearMonthDateRange(YearMonth yearMonth) {
    Objects.requireNonNull(yearMonth);
    startDate = yearMonth.atDay(1);
    endDate = yearMonth.atEndOfMonth();
  }

  @Override
  public LocalDate startDate() {
    return startDate;
  }
```

```
  @Override
  public LocalDate endDate() {
    return endDate;
  }

  YearMonth yearMonth() {
    return YearMonth.from(startDate);
  }

  // ...
}
```

Mimo, że oba zakresy określają datę początkową i końcową nawet przechowując te same pola, to różnią się sposobem ich stworzenia, dodatkowym zachowaniem czy tym, co będzie można z nimi zrobić w danej domenie. Takie różne obiekty wartości precyzyjnie określają pewien koncept domenowy — czasem ekspert domenowy posługuje się pojęciem całego miesiąca, a czasem pewnego okresu niezwiązanego z miesiącami — i to zostało odwzorowane w kodzie. Takiego rozdzielenia zachowań i odpowiedzialności nie dało by się uzyskać jednym obiektem wartości. Oczywiście oba te obiekty wartości mogą mieć wspólny interfejs (tu: `DateRange`), bo w różnych miejscach systemu interesować klienta będzie jedynie to, że zakres ma początek i koniec. Takie wspólne traktowanie obiektu wartości musi być jednak świadomie zaprojektowane, bo mimo tych samych cech może się okazać, że błędnie jest wykorzystywać te obiekty jako tożsame poprzez interfejs.

Podsumowując, problemy związane z obiektami wartości podobnych typów należy rozwiązywać odpowiadając na dwa pytania: czy zachowania podobnych obiektów wartości różnią się oraz czy podobne obiekty wartości są inaczej traktowane w domenie. Jeżeli na co najmniej jedno z tych pytań odpowie się twierdząco — to należy stworzyć różne obiekty wartości. Jeżeli nie — to należy stworzyć ten sam obiekt wartości.

4.2 Implementacja

Właściwa implementacja obiektu wartości to zagwarantowanie jego niemutowalności (niezmienności). Dzięki temu, utrzymanie spójności obiektu wartości polega jedynie na walidacji parametrów wejściowych podczas tworzenia obiektu. Nigdy później nie trzeba przejmować się utrzymywaniem spójnego stanu obiektu wartości, gdyż nie da się go zmienić. Jedyny sposób zmiany obiektu wartości, to jego całkowita wymiana na nowy, inny obiekt o innej wartości. Taka implementacja obiektu wartości gwarantuje

udostępnianie jego zachowania bez skutków ubocznych, gdyż zmiana stanu wewnętrznego nie jest możliwa. Jest to też zgodne z, ostatnio popularnym, programowaniem funkcyjnym. Można powiedzieć, że operowanie niemutowalnymi obiektami wartości jest cechą wspólną programowania obiektowego oraz funkcyjnego.

4.2.1 Niemutowalna klasa

Implementacja niemutowalnej klasy w Javie jest nietrywialna i polega na tym, że klasa nie może pozwolić na zmianę raz przypisanych wartości pól. Warto tu zwrócić uwagę na obiekty mutowalne wchodzące w skład takiej klasy. Żeby nie pozwolić na późniejszą zmianę stanu przy konstrukcji takiego obiektu należy skopiować taki obiekt mutowalny i jego używać jako składową klasy. Również przy zwracaniu takiej składowej, należy zwrócić jej kopię, a nie wykorzystywaną referencję. W związku z tymi wytycznymi, klasa niemutowalna posiada następujące cechy:

- wszystkie jej pola są prywatne i finalne;
- nie udostępnia metod zmieniających stan (np. setterów);
- ustawienie pól dzieje się wyłącznie w konstruktorze — parametry niemutowalne są bezpośrednio przypisywane, mutowalne są kopiowane;
- metody zwracające składowe niemutowalne po prostu je zwracają;
- metody zwracające składowe mutowalne zwracają kopie tych składowych;
- jest finalna.

Poniżej implementacja referencyjna klasy niemutowalnej:

Listing 4.9: Wzór klasy niemutowalnej

```
import com.google.common.collect.ImmutableList;
import com.google.common.collect.ImmutableSet;

final class ImmutableValueObject {
  private final ImmutableClass immutableField;
  private final MutableClass mutableField;
  private final Set<ImmutableClass> immutableClassSet;
  private final List<MutableClass> mutableClassList;

  ImmutableValueObject(
      ImmutableClass immutableField,
      MutableClass mutableField,
```

```
      Set<ImmutableClass> immutableClassSet,
      List<MutableClass> mutableClassList) {
    this.immutableField = immutableField;
    this.mutableField = MutableClass.copy(mutableField);
    this.immutableClassSet = ImmutableSet
          .copyOf(immutableClassSet);
    this.mutableClassList = ImmutableList
          .copyOf(mutableClassList);
  }

  ImmutableClass immutableField() {
    return immutableField;
  }

  MutableClass mutableField() {
    return MutableClass.copy(mutableField);
  }

  Set<ImmutableClass> immutableClassSet() {
    return immutableClassSet;
  }

  List<MutableClass> mutableClassList() {
    return mutableClassList;
  }

  // ...
}
```

W konstruktorze obiekt mutowalnej klasy został skopiowany. Oznacza to, że aby dało się utworzyć klasę niemutowalną z mutowalnym składnikiem to takie kopiowanie musi być zaimplementowane w klasie mutowalnej. Takie kopiowanie jest również wykonywane w metodzie dostępowej, żeby chronić posiadaną referencję do obiektu mutowalnego.

Podobnie stało się z kolekcjami, które są bardzo często mutowalne. Tu jednak z pomocą przychodzi biblioteka Guava [2017] i jej niemutowalne kolekcje, których konstruktory kopiujące zostały użyte w konstruktorze. Dzięki temu, że mutowalne parametry wejściowe zostały zamienione na niemutowalne pola, nie ma potrzeby wykonywania dodatkowej kopii w metodach dostępowych.

Niemutowalne kolekcje niemutowalnych obiektów są doskonale niemutowalne. Problem pojawia się z niemutowalnymi kolekcjami mutowalnych obiektów lub, bardziej generalnie, z mutowalnymi składowymi zawierającymi inne mutowalne składowe. Aby osiągnąć perfekcyjną niemutowalność w takim przypadku, należałoby wykonać głęboką kopię (ang. deep copy) całej struktury mutowalnego obiektu. Jest to rozwiązanie kosztowne, a czasem wręcz

niemożliwe do wykonania. W większości sytuacji, zupełnie wystarczające jest wykonanie płytkiej kopii (ang. shallow copy) i to powinno być domyślne zachowanie. Tylko wtedy, gdy płytka kopia będzie generować problemy, można postarać się zaimplementować kopiowanie głębokie.

Biorąc pod uwagę dodatkowe komplikacje wynikające z faktu komponowania mutowalnego obiektu, należy starać się tak projektować obiekty niemutowalne, aby posiadały jak najmniej pól mutowalnych, jak to tylko możliwe.

4.2.2 Metody porównujące

Obiekt wartości cechuje to, że „można go porównać z innymi za pomocą równości wartości" [Vernon, 2013]. Sprawdzanie równości obiektu w języku Java jest gwarantowane przez metodę `equals(Object)`, która stanowi interfejs klasy `Object`, a więc każdej klasy w języku Java, gdyż one wszystkie niejawnie dziedziczą tę klasę.

Właściwa implementacja metody `equals()` (oraz korespondującej z nią metody `hashCode()`) polega na użyciu wszystkich pól klasy obiektu wartości, ponieważ wszystkie te pola są niezmienialne w obrębie tej klasy. Warto jedynie wziąć pod uwagę pola, które mogą mieć wartość `null`, gdyż należy je dodatkowo obsłużyć. W związku z tym, warto także minimalizować liczbę opcjonalnych pól. Poniżej implementacja referencyjna:

Listing 4.10: Wzór implementacji metod porównujących w obiekcie wartości

```
final class ValueObject {
  private final Object notNullField;
  private final Object nullableField;

  ValueObject(Object notNullField, Object nullableField) {
    this.notNullField = Objects.requireNonNull(notNullField);
    this.nullableField = nullableField;
  }

  // ...

  @Override
  public boolean equals(Object o) {
    if (this == o) {
      return true;
    }
    if (!(o instanceof ValueObject)) {
      return false;
    }

    ValueObject that = (ValueObject) o;
```

```
    if (!notNullField.equals(that.notNullField)) {
      return false;
    }
    return nullableField != null ?
        nullableField.equals(that.nullableField) :
        that.nullableField == null;
  }

  @Override
  public int hashCode() {
    int result = notNullField.hashCode();
    result = 31 * result + (nullableField != null ?
          nullableField.hashCode() : 0);
    return result;
  }
}
```

Czasem utożsamianie metody porównującej z metodą `equals()` może być niepożądane. Zarówno w przykładowym projekcie DDD [Citerus] jak i [Sobótka, 2013] autorzy oprócz metody `equals()` zamieszczają inne metody porównawcze. W tym pierwszym w obiektach wartości znajdziemy metodę `sameValueAs()`, a w tym drugim metodę `sameAs()`, która oprócz parametru porównawczego przyjmuje pewną deltę — akceptowalny błąd porównania. Takie metody mogą być bardziej odpowiednie semantycznie w domenie.

Inną kwestią jest wykorzystanie metody `hashCode()` (jednoznacznie związanej z `equals()`) w kolekcjach zapewniających unikalność zawieranych elementów, takich jak `Set` czy `Map`. Unikalność elementów, będących obiektami wartości zawartymi w takich kolekcjach, może wymagać innej metody na sprawdzanie tej unikalności, a innej na porównanie jej z innymi obiektami. Przykładem może być obiekt wartości opisujący dochód w danym miesiącu roku. Przy tworzeniu zbioru (ang. set) takich obiektów nie można byłoby dopuścić do powstania zduplikowanych miesięcy z różnymi dochodami. Można to zaimplementować w poniższy sposób:

Listing 4.11: Przykład różnych metod na sprawdzanie unikalności obiektu i porównywanie wartości

```
class MonthlyRevenue {
  private final YearMonth yearMonth;
  private final BigDecimal revenue;

  // ...

  @Override
  public boolean equals(Object o) {
    if (this == o) {
      return true;
    }
    if (!(o instanceof MonthlyRevenue)) {
      return false;
    }

    MonthlyRevenue other = (MonthlyRevenue) o;

    return yearMonth.equals(other.yearMonth);
  }

  @Override
  public int hashCode() {
    return yearMonth.hashCode();
  }

  boolean sameAs(MonthlyRevenue other) {
    if (other == null) {
      return false;
    }
    if (!yearMonth.equals(other.yearMonth)) {
      return false;
    }
    return revenue.compareTo(other.revenue) == 0;
  }
}
```

Metody `equals()` i `hashCode()` zostały zaimplementowane tak, aby brały pod uwagę jedynie dany miesiąc. Można je potraktować jako metody „techniczne" służące jedynie zapewnieniu unikalności w zawierających tę klasę kolekcjach.

Metoda realizująca porównywanie dwóch obiektów wartości to metoda `sameAs()` i to jej należy używać realizując domenową potrzebę porównania dwóch obiektów wartości `MonthlyRevenue`. Warto zwrócić uwagę, że metoda ta porównując dochód o typie `BigDecimal` nie wykorzystuje jego metody

`equals()`, a `compareTo()`. Jest to zgodne z logiką domenową porównania dwóch wartości dochodów, gdyż w domenie wartości np. `10.0` i `10.00` są sobie równe, mimo iż posiadają inną skalę, co zgodnie z implementacją metody `BigDecimal.equals()` byłoby nierówne (zwróciło by `false`).

Podsumowując, polecam implementację metod `equals()` i `hashCode()` wykorzystującą wszystkie pola składowe obiektu wartości, chyba że, w celu zapewnienia unikalności tych obiektów w przechowujących je kolekcjach, wykorzystać należy jedynie ich podzbiór. W tym wypadku powinno się metodę `equals()` traktować jako „techniczną", a co za tym idzie należy stworzyć inną metodę realizującą porównywanie obiektów wartości — metodę `sameAs()`.

Rozdział 5

Agregaty

Najlepszą i najbardziej zwięzłą definicję agregatu wprowadził twórca DDD Eric Evans [2004]:

> Agregat jest zbiorem skojarzonych obiektów, które na potrzeby zmiany danych traktujemy łącznie. Każdy agregat posiada korzeń oraz granicę. Ta ostatnia definiuje wszystko, co znajduje się wewnątrz agregatu. Korzeń jest tylko pojedynczą określoną encją zawartą w agregacie. Jest ona jedynym elementem agregatu, do którego mogą odwoływać się obiekty z zewnątrz. Tym niemniej obiekty wewnątrz granicy mogą mieć referencje do siebie nawzajem. Encje różne od korzenia posiadają lokalną tożsamość, jednak jest ona rozróżnialna jedynie wewnątrz agregatu, ponieważ nie może jej widzieć żaden zewnętrzny obiekt spoza kontekstu encji będącej korzeniem.

Z powyższego cytatu oraz z dodatkowych informacji o agregacie zawartych w [2004] można stwierdzić, że:

- agregat to graf obiektów składający się z korzenia (określonej encji) i granicy, będący spójną jednostką zmiany [Sobótka, 2012];
- jednym z zadań agregatu jest utrzymywanie spójności (utrzymywanie niezmienników);
- można odnosić się jedynie do korzeni agregatów;
- można modyfikować tylko jeden agregat w obrębie jednej transakcji
- korzenie agregatów muszą mieć unikalne ID w obrębie całego systemu;
- ID obiektów wewnątrz agregatu, nie będących korzeniem, może być unikalne jedynie lokalnie w obrębie agregatu.

5.1 Skład agregatu

Agregat bywa czasem mylnie utożsamiany z korzeniem agregatu. Wpływa na to fakt, że agregat nie może istnieć bez korzenia, a sam korzeń (pojedyncza encja) może być całym agregatem. Agregaty o wielkości jednego obiektu są jak najbardziej dopuszczalne, a w świetle zasady, że agregaty powinny być jak najmniejsze — są optymalne pod względem wielkości. Jednak to nie wielkość agregatu świadczy o jego optymalności, a to w jaki sposób oddaje on rzeczywistość biznesową zmieniających się razem obiektów.

Definiowanie składu agregatu (inaczej: określanie jego granicy) jest jednym z najtrudniejszych wyzwań przy stosowaniu podejścia DDD. Istnieje wiele sposobów na odpowiednie wyznaczanie granicy, ale żaden nie daje pewności, że wyznaczenie to jest optymalne.

Jedną z kluczowych zasad jest myślenie o obiektach wchodzących w skład agregatu nie jak o grafie obiektów, gdzie jeden obiekt zawiera drugi, ale jak o zbiorze obiektów, które muszą być razem zmienione, żeby nie utraciły swojej spójności. Przykładem może być system zarządzający sklepami i produktami. Sklep ma jakiegoś kierownika, adres i zawiera pewne produkty. Produkty mają ceny detaliczne i hurtowe. Istnieje pokusa zamodelowania agregatu mającego jako korzeń sklep, gdyż sklep jest korzeniem tej kompozycji obiektów. W świetle takiej granicy agregatu każda zmiana ceny detalicznej produktu spowoduje modyfikację agregatu sklep, mimo iż sklepu „nie obchodzi" cena konkretnego produktu, tylko czy jest on dostępny do sprzedaży i w jakiej ilości. Sklep powinien zostać zamodelowany jako osobny agregat i produkt również. Idąc tym tokiem myślenia można się zastanowić, czy produkt „jest zainteresowany" zmianą ceny detalicznej, czy to też należy rozdzielić na dwa agregaty. W danym przykładzie cena detaliczna jest zależna od ceny hurtowej — nie może spaść poniżej jej poziomu. W związku z tym, cena detaliczna będzie się zmieniała w zależności od produktu, a więc razem z nim. Wobec tego powinna zostać ujęta w obrębie granicy agregatu „produkt".

Innym, uproszczonym sposobem ustalania granicy agregatu jest zasada tzw. „usuwania kaskadowego" [Lerman i Smith, 2014]. Polega ona na sprawdzaniu, czy obiekt zawierany ma sens sam w sobie, czy też należy go usunąć kaskadowo przy usuwaniu obiektu go zawierającego. Korzystając z wcześniejszego przykładu możemy uznać, że cena detaliczna sama w sobie nie ma sensu istnienia bez produktu, który ją zawiera. Możemy też dojść do wniosku, że usunięcie sklepu nie powinno kaskadowo usunąć produktów, bo dany produkt może istnieć i bez zawierającego go sklepu, a więc agregat „produkt" jest jak najbardziej osobnym agregatem od agregatu „sklep".

Ustanawianie jak najmniejszych agregatów jest dobrą zasadą. Łatwiej jest zmodyfikować agregat mały niż duży. Łatwiej też jest na nim operować —

wczytywać go do pamięci i modyfikować równolegle. Nie występują problemy związane z długimi transakcjami, gdyż transakcjami objęte są jedynie precyzyjne zmiany w jednym agregacie. Oczywiście nie zawsze da się stworzyć małe agregaty — czasem obiekty tak bardzo zależą od siebie, że mogą nie mieć sensu występując osobno.

5.2 Utrzymywanie niezmienników

Głównym celem grupowania obiektów w agregaty jest łatwiejsze utrzymywanie spójności tych obiektów wobec siebie. Agregaty mają za cel stać na straży swoich niezmienników. Niezmiennik to reguła biznesowa, która musi być zawsze spełniona. Niezmiennikiem może być przykładowo to, że każdy użytkownik w systemie posiada poprawny e-mail. Nie można stworzyć użytkownika bez e-maila, nie można edytować użytkownika usuwając e-mail. Nie można tak edytować e-maila, żeby był on niepoprawny. Innym przykładem niezmiennika może być np. reguła mówiąca, że zamówienie może mieć maksymalnie dziesięć różnych produktów. Tak zamodelowany agregat „zamówienie" spowoduje, że jakakolwiek próba dodania jedenastego produktu nigdy się nie uda.

Utrzymanie niezmienników w agregacie jest możliwe tylko wtedy, gdy klienty agregatu komunikują się wyłącznie przez korzeń agregatu wykonując zarówno polecenia, jak i zapytania. W momencie, gdy „wycieknie" do kodu klienckiego jakakolwiek mutowalna referencja do obiektu lokalnego względem agregatu przestanie być możliwe utrzymanie niektórych niezmienników, powodując utratę spójności skutkującą potencjalnymi błędami w dalszym projektowaniu oraz działaniu aplikacji.

Odnoszenie się do obiektów lokalnych z zewnątrz agregatu powinno się odbywać na podstawie lokalnego ID danego obiektu podanego jako parametr metody, chyba że obiektów lokalnych danego typu jest niewiele — wtedy można stworzyć osobne metody dla takich obiektów. Zgodnie z zasadą przedstawioną w poprzednim paragrafie, wartością zwracaną z metod-zapytań wywołanych na korzeniu agregatu nie może być mutowalna referencja do lokalnego obiektu. Bezpiecznymi wartościami zwracanymi w takim wypadku będą: wszystkie prymitywne wartości, struktury służące transferowi danych (tzw. DTO [Fowler, 2003a]) lub obiekty typu Data [Sobótka, 2013] stworzone z prawdziwych mutowalnych obiektów lokalnych, a także niemutowalne obiekty wartości (nawet jeżeli efektywnie są jedynie obiektami lokalnymi danego agregatu). Używając takich wartości agregat nie rozszczelnia się.

Dodatkową zaletą takiego rozwiązania jest silna enkapsulacja obiektów lokalnych w obrębie agregatu, co niesie ze sobą wszystkie korzyści z enkap-

sulacji, takie jak dowolność zmiany implementacji czy luźniejsze powiązania między klasami. Wspierającą to podejście regułą, którą chciałbym wprowadzić, jest tworzenie klas lokalnych z domyślnym dostępem pakietowym. Przy dobrym zamodelowaniu pakietów, takim że każdy agregat jest w odrębnym pakiecie, kompilator nie pozwoli na użycie takiej klasy lokalnej przez kod kliencki w innym pakiecie. W związku z tym, że obiekty lokalne są zenkapsulowane i mogą być niedostępne dla jakiegokolwiek kodu spoza obrębu agregatu dodałbym jeszcze jedną charakterystykę agregatu: agregat powinien zarządzać cyklem życia obiektów lokalnych, tzn. powinien je tworzyć, modyfikować i usuwać w ramach potrzeby. Uznając, że będą się one mieściły jedynie w pakiecie danego agregatu i miały dostęp pakietowy — to i tak jedyne rozsądne miejsce na tego typu operacje.

W agregatach można odnosić się do innych agregatów tylko poprzez ich korzenie. Warto stosować się przy tym do zasady wprowadzonej przez Vaughna Vernona [2013] — należy odnosić się do innych agregatów przez ID ich korzenia. Stosowanie się do tej zasady zmniejsza możliwość popełnienia błędu zmiany więcej niż jednego agregatu w ramach jednej transakcji. W sytuacjach, gdzie taka zmiana jest potrzebna — należy użyć tzw. „ostatecznej spójności" (ang. eventual consistency). Jeżeli takie zmiany zachodzą często to należy rozważyć zmianę tych agregatów i połączenie ich w jeden, ale trzeba wtedy wziąć pod uwagę wielkość takiego agregatu.

5.3 Implementacja

Przykład prawidłowej implementacji korzenia agregatu znajduje się poniżej:

Listing 5.1: Przykład poprawnej implementacji korzenia agregatu

```
class Client {
  private final ClientId clientId;
  private String name;
  private Map<String, Brand> brands;
  private Set<DiscountId> discounts;

  Client(String name) {
    clientId = new ClientId();
    this.name = name;
    brands = new HashMap<>();
    discounts = new HashSet<>();
  }

  void addDiscount(DiscountId discountId) {
```

```
    discounts.add(discountId);
  }

  Set<DiscountId> discounts() {
    return ImmutableSet.copyOf(discounts);
  }

  void addBrand(String name,
                PopularityIndex popularityIndex) {
    if (brandExists(name)) {
      throw new BrandAlreadyExistsException();
    }
    brands.put(name, Brand.create(name, popularityIndex));
  }

  private boolean brandExists(String name) {
    return brands.get(name) != null;
  }

  void changeBrandPopularity(
      String name, PopularityIndex newPopularityIndex) {
    Brand brand = brands.get(name);
    brand = brand.changePopularity(newPopularityIndex);
    brands.put(name, brand);
  }

  BrandDto brand(String name) {
    Brand brand = brands.get(name);
    return brand.toDto();
  }

  ClientDto toDto() {
    return new ClientDto(
        clientId, name, brands(), discounts());
  }

  private Set<BrandDto> brands() {
    return brands.values()
                 .stream()
                 .map(Brand::toDto)
                 .collect(Collectors.toSet());
  }

  // ...
}
```

Korzeń agregatu (i efektywnie cały agregat) `Client` składa się z globalnie unikalnego ID, nazwy, lokalnych obiektów — marek (ang. brands) i kolekcji referencji do innych agregatów — zniżek (ang. discounts).

Referencje do agregatów zniżek utrzymywane są jedynie poprzez identyfikatory tych agregatów. Można zauważyć, że możliwości wykorzystania agregatu `Discount` w agregacie `Client` są znikome, co skutecznie uniemożliwia jakąkolwiek zmianę tego agregatu w tym miejscu. Ponadto, referencje są obiektami wartości, więc można je bezpiecznie zwracać na żądanie. Ważne jedynie, żeby zwrócić uwagę na odpowiednie opakowanie kolekcji, która jest mutowalna.

Obiekty `Brand` to obiekty lokalne agregatu `Client`. Są to obiekty wartości o unikalnej nazwie w obrębie agregatu. Ich cyklem życia zarządza korzeń agregatu: są tworzone jedynie przez korzeń agregatu w momencie, gdy kod kliencki wywołuje metodę `addBrand()` na korzeniu. Istnieją tylko i wyłącznie w ramach agregatu, klasa `Brand` ma zasięg pakietowy. W celu modyfikacji marki postępuje się analogicznie jak przy jej dodaniu — przekazuje się jedynie odpowiednie parametry do metody korzenia agregatu. Odnoszenie się do konkretnego obiektu `Brand` odbywa się przez jego lokalne ID, tu: nazwę marki. Przy wywoływaniu konkretnych metod na obiekcie `Brand` korzeń agregatu — `Client` — staje się jedynie opakowaniem tego obiektu, delegując wszelkie operacje do konkretnego obiektu `Brand`.

Wartościami zwracanymi z metod-zapytań są jedynie obiekty, które nie wpłyną na agregat `Client`. Są to jedynie obiekty niemutowalne (ID, nazwa, referencje do agregatów zniżek) albo obiekty typu DTO stworzone z konkretnych obiektów domenowych. Dzięki temu, widać tu pełną enkapsulację implementacji agregatu. Implementacja obiektów wewnętrznych (tu: marek) może zostać całkowicie wymieniona i nie będzie to miało żadnego wpływu na kod kliencki agregatu.

Pragnę tu także zauważyć, że przedstawiona implementacja zawiera bezpośrednio kolekcje `Map` i `Set`. Służą one jedynie lepszej prezentacji przykładu poprawnej implementacji agregatu i hermetyzacji implementacji. Wzorowo, zamiast tych kolekcji powinny być użyte klasy opakowujące kolekcje (zob. 4.1.1).

Rozdział 6

Fabryki

Fabryki to metody lub klasy służące do tworzenia obiektów. Tworzenie obiektów to pewien detal implementacyjny, który może nie mieć odzwierciedlenia w modelowanej dziedzinie. Jednak mimo to, że fabryki mogą nie wyrażać modelu biznesowego jako takiego, jak najbardziej posiadają odpowiedzialności warstwy domenowej i tam też należą [Evans, 2004].

Motywacją do definiowania fabryki jako osobnego bytu jest fakt, że konstruowanie obiektów bywa skomplikowaną operacją, która nie ma dużo wspólnego z późniejszym używaniem tego obiektu, z jego późniejszą odpowiedzialnością. Można powiedzieć, że w pewien sposób skomplikowana kreacja łamie zasadę pojedynczej odpowiedzialności (ang. SRP [Martin, 2003, s. 95]). Rozwiązaniem zgodnym z tą zasadą byłoby przeniesienie tej logiki i odpowiedzialności na klienta, jednakże wtedy klient nie tylko staje się bardziej skomplikowany, ale także musi mieć wiedzę w jaki sposób utworzyć obiekt domenowy. Ponadto, klient używający bezpośrednio konstruktorów obiektów domenowych mocno wiąże się z istniejącą ich implementacją co znacząco utrudnia wprowadzanie kolejnych zmian oraz refaktoring.

W związku z powyższym, Evans [2004] stawia dwa wymagania przed dobrą fabryką:

1. może tworzyć jedynie poprawne i spójne obiekty wymuszając spełnienie wszystkich ich niezmienników;
2. powinna zwracać typy abstrakcyjne (np. interfejsy) zamiast konkretnych klas.

6.1 Rodzaje i miejsce fabryk

Możemy wyróżnić cztery rodzaje fabryk:

1. prosty konstruktor;
2. metoda wytwórcza [Gamma et al., 1994] w korzeniu agregatu tworząca agregat lub jego elementy;
3. metoda wytwórcza umieszczona w korzeniu agregatu tworząca inny agregat lub obiekt, który nie jest częścią tworzącego go agregatu;
4. osobna klasa.

6.1.1 Konstruktor

Pomimo słusznej krytyki zbyt powszechnego używania konstruktorów przez klienty, Evans [2004] zauważa, że w kilku specyficznych przypadkach to użycie ma sens:

- konstruowana klasa jest typem (np. obiektem wartości), którego nie wykorzystuje się polimorficznie i nie należy do żadnej interesującej hierarchii klas;
- klient przykłada wagę do konkretnej implementacji, np. wybierając odpowiednią strategię;
- konstrukcja obiektu nie jest skomplikowana.

Konstruktory przede wszystkim powinny być bardzo proste. Jeżeli przenika do nich jakakolwiek większa wiedza lub walidacja przekracza sprawdzenie czy referencja przekazana przez parametr jest pusta (tzn. równa `null`) — należy użyć innego rodzaju fabryki, np. metody wytwórczej.

6.1.2 Metoda wytwórcza w korzeniu tworzonego agregatu

Ten rodzaj fabryki jest dobrą opcją domyślną przy tworzeniu każdej nowej fabryki. Jak zostało wspomniane w rozdziale 5 („Agregaty") agregat (jego korzeń) jest odpowiedzialny za tworzenie swoich obiektów lokalnych. Fabryka obiektów lokalnych musiałaby być oczywiście niedostępna dla kodu klienckiego, ale mogłaby dobrze działać w obrębie agregatu (mogłaby być zaimplementowana z dostępem pakietowym). Ten rodzaj fabryk sprawdzi się również przy tworzeniu całego agregatu, jeżeli nie jest on bardzo skomplikowany. Statyczna metoda wytwórcza [Bloch, 2008, temat 1.] pozwoli na zwrócenie odpowiedniej abstrakcji klientowi, a umieszczenie jej wewnątrz korzenia agregatu sprawia, że zapewnienie jej zachowania prawidłowych niezmienników staje się łatwe, gdyż metody wymuszające niezmienniki znajdują się w tej samej klasie.

Zdecydowaną wadą tego podejścia jest mała izolacja tworzenia obiektów od używania ich, co przyświecało idei powstania pojęcia fabryk. To sprawia, że ten rodzaj fabryk sprawdza się jedynie przy małych i nieskomplikowanych agregatach.

6.1.3 Metoda wytwórcza w korzeniu innego agregatu

To najrzadziej spotykana forma fabryki. Jej zalety są takie same jak każdej innej metody wytwórczej, jednakże nie posiada ona wiedzy o niezmiennikach tworzonego obiektu, który poza stworzeniem, jest od niej niezależny. W związku z tym, proces tworzenia tego obiektu nie może być skomplikowany.

Ten rodzaj fabryki może być stosowany, gdy tworzenie nowego obiektu silnie zależy od tworzonego ją obiektu. Przykładem może być stworzenie nowego agregatu-zamówienia z agregatu-użytkownik. Stworzone zamówienie może od razu mieć informacje o preferencjach użytkownika, np. może mieć ustawiony przez agregat-użytkownik domyślny adres wysyłki.

Wad tego podejścia jest kilka. To ciągle agregat, mający inne odpowiedzialności, zajmuje się tworzeniem obiektu (tu: innego agregatu), który nie posiada obiektu tworzonego. Łatwo pomieszać odpowiedzialności i związać tworzący agregat z tworzonym obiektem, mimo iż takie powiązanie może nie mieć sensu poza samym aktem kreacji obiektu.

Trudno znaleźć wytłumaczenie wprowadzenia takiego rodzaju fabryki przez Evansa [2004]. Z łatwością tą samą funkcjonalność, czyli wykorzystanie zawartości lub zachowania innego agregatu, można uzyskać tworząc dedykowaną klasę fabryki, która jako jedną z zależności tworzenia nowego obiektu przyjmie wspomniany agregat. Tym samym unika się dokładania odpowiedzialności tworzenia jednego agregatu zupełnie innemu agregatowi, co zmniejsza złożoność przypadkową.

6.1.4 Osobna klasa

Fabryka jako osobna klasa to jedyne rozsądna opcja, gdy tworzenie obiektów zależy od czynników zewnętrznych, takich jak inne agregaty, usługi, źródła danych, zmienne środowiskowe czy zmienne aplikacji.

Mogą one być implementowane zarówno jako zwykłe obiekty (ang. POJO — Plain Old Java Object [Fowler et al., 2000]) tworzone w tych miejscach, gdzie są potrzebne lub np. jako ziarna (ang. beans) Springa o zasięgu typu singleton. Obie implementacje dają możliwości wstrzyknięcia potrzebnych zależności do klasy wytwórczej, która potem może z nich skorzystać w znany

sobie sposób. W ten sposób fabryki zyskują dodatkowe możliwości nie komplikując kodu klienckiego lub tworzonego obiektu.

Samodzielna fabryka jest najczęściej odpowiedzialna za tworzenie całego agregatu. Powinna znajdować się w pakiecie razem z tworzonym obiektem — umożliwia to dostęp do niektórych niedostępnych publicznie metod lub pól tworzonego obiektu. Dodatkowo, taka fabryka, oprócz publicznie dostępnej metody tworzącej cały agregat, może posiadać metody dostępne pakietowo, które budują lokalne encje lub obiekty wartości używane przez korzeń budowanego agregatu.

Jedyną wadą osobnych fabryk jest to, że trzeba poświęcić więcej czasu i linii kodu na ich napisanie, niż na poprzednie rodzaje fabryk. Ilość kodu nie jest jednak znacząco większa, bo w najprostszym porównaniu różnica między metodą wytwórczą w agregacie, a konstruktorem wynosi dwie linijki: otwierającą i zamykającą definicję osobnej klasy.

6.2 Implementacja

Rozważając jak powinna wyglądać implementacja fabryki wzięto pod uwagę dwa główne rodzaje fabryk: metodę wytwórczą w korzeniu tworzonego agregatu oraz osobną klasę fabryki.

Poniżej znajdują się przykłady fabryk jako metody wytwórcze agregatu:

Listing 6.1: Przykłady fabryk jako metody wytwórcze agregatu

```
public interface City {

  public static City create(String name,
                            Population population) {
    Objects.requireNonNull(name);
    Objects.requireNonNull(population);
    CitySize citySize = CitySize.basedOn(population);
    switch (citySize) {
      case BIG_CITY:
        return new BigCity(name, population);
      case TOWN:
        return new Town(name, population);
      case VILLAGE:
        return new Village(name, population);
      default:
        throw new AssertionError();
    }
  }

  // ...
```

```
}

class BigCity implements City {

  private final String name;
  private final Population population;
  private final Set<Shop> shops;

  BigCity(String name, Population population) {
    this.name = name;
    this.population = population;
    shops = new HashSet<>();
  }

  void addShop(String name, Owner owner, Address address,
               Capacity capacity) {
    Shop shop = createShop(name, owner, address, capacity);
    shops.add(shop);
  }

  private Shop createShop(String name,
                          Owner owner,
                          Address address,
                          Capacity capacity) {
    Objects.requireNonNull(name);
    Objects.requireNonNull(owner);
    Objects.requireNonNull(address);
    Objects.requireNonNull(capacity);
    if (address.isNotInCity(this.name)) {
      throw new ShopInDifferentCityException();
    }
    return Shop.create(name, owner, address, capacity);
  }

  // ...
}
```

W interfejsie `City` znajduje się statyczna metoda wytwórcza, która na podstawie rozmiaru populacji i powstającego z niego obiektu rozmiaru miasta `CitySize` (typu enum) tworzy swoje odpowiednie implementacje. Zgodnie z założeniami fabryk — metoda ta zwraca interfejs `City`. Metoda tworząca nie jest bardzo skomplikowana i łatwo się z niej korzysta.

Inna metoda wytwórcza znajduje się w klasie `BigCity`. Klasa ta, będąc korzeniem agregatu, przy dodawaniu sklepu, czyli swojego obiektu lokalnego, tworzy go sprawdzając niezmienniki. Tu metoda wytwórcza nie jest statyczna, wymaga istnienia wcześniej korzenia agregatu typu `BigCity`. Jest to jednak zamierzone działanie — można stworzyć sklep w systemie jedynie

dodając go do istniejącego miasta.

W powyższym rozwiązaniach, można zauważyć, że w interfejsie `City` obiekt `CitySize` powstaje jedynie na potrzeby wybrania odpowiedniej implementacji do stworzenia. Jest on potrzebny jedynie w procesie tworzenia — co sugeruje, że lepszym sposobem tworzenia agregatów `City` mogłaby być fabryka jako osobna klasa:

Listing 6.2: Przykład fabryki jako osobnej klasy

```
public class CityFactory {

  public City create(String name, Population population) {
    Objects.requireNonNull(name);
    Objects.requireNonNull(population);
    CitySize citySize = CitySize.basedOn(population);
    switch (citySize) {
      case BIG_CITY:
        return new BigCity(name, population);
      case TOWN:
        return new Town(name, population);
      case VILLAGE:
        return new Village(name, population);
      default:
        throw new AssertionError();
    }
  }
}
```

Dzięki zastosowaniu osobnej klasy logika tworzenia danego agregatu została oddzielona od logiki zachowań danego agregatu. Niezmienniki dot. tworzenia agregatu mogą zostać umieszczone poza agregatem. Sprzyja to również spełnianiu zasady SRP, gdzie ta pojedyncza odpowiedzialność klasy fabryka — to tworzenie obiektów.

Wykorzystanie osobnej klasy fabryki to również dodatkowe możliwości związane z użyciem zewnętrznych zależności. W prosty sposób do takiej fabryki można wstrzyknąć potrzebne zależności do utworzenia agregatu sprawdzające jakieś dodatkowe warunki biznesowe lub dostarczające dodatkowe dane do umieszczenia w agregacie. Przykład takiej fabryki poniżej:

Listing 6.3: Przykład fabryki z zewnętrznymi zależnościami

```
public class ShopFactory {
  private final CityRepository cityRepository;
  private final ShopIdGenerator shopIdGenerator;

  public ShopFactory(CityRepository cityRepository,
                     ShopIdGenerator shopIdGenerator) {
    this.cityRepository = cityRepository;
    this.shopIdGenerator = shopIdGenerator;
  }

  private Shop createShop(String name,
                          Owner owner,
                          Address address,
                          Capacity capacity) {
    Objects.requireNonNull(name);
    Objects.requireNonNull(owner);
    Objects.requireNonNull(address);
    Objects.requireNonNull(capacity);
    checkIfExistsCityForAddress(address);
    ShopId shopId = shopIdGenerator.nextId();
    return Shop.create(
        shopId, name, owner, address, capacity);
  }

  private void checkIfExistsCityForAddress(Address address) {
    String cityName = address.cityName();
    if (!cityRepository.existsByName(cityName)) {
      throw new ShopAddressInNotExistingCityException(
          cityName);
    }
  }
}
```

Jak wynika z przykładu, do stworzenia, niezależnego teraz, agregatu-sklep potrzebujemy sprawdzić, czy istnieje w systemie już jakieś miasto, które jest zawarte w adresie nowego sklepu. Dodatkowo, ID sklepu jest generowane zewnętrznie przez `ShopIdGenerator`, który jest wstrzyknięty jako zależność fabryki. Zarówno logika związana z adresem, jak i ID sklepu dotyczy wyłącznie procesu tworzenia nowego sklepu. W związku z tym, umieszczenie jej w dedykowanej fabryce zwiększa czytelność kodu, poprzez jasne oddzielenie logiki związanej z tworzeniem od logiki związanej z używaniem agregatu.

Rozdział 7

Repozytoria

W aplikacjach, które nie działają wyłącznie w pamięci, czyli w przeważającej ich większości, obiekty mają dwa dodatkowe zmiany stanów w ich cyklu życia: utrwalanie danych (ang. persistence) — zapis stanu obiektu poza aplikacją oraz rekonstytucja — ponowne stworzenie obiektu i załadowanie jego stanu do pamięci programu. Do realizacji tych operacji w DDD stosuje się repozytoria.

Repozytorium, podobnie jak fabryka, nie jest bytem stricte domenowym, jako że nie istnieje w biznesowym języku klienta. Niemniej jednak, wprowadzenie repozytorium jako pewnej koncepcji, która opakowuje bardzo techniczną logikę zapisu i odczytu danych, pozwala w warstwie domeny skupić się na tym co istotne przy jej modelowaniu, zamiast wgłębiać w techniczne detale implementacji zapisu lub odczytu danych z bazy danych, plików czy innych systemów utrwalania danych. Repozytorium stanowi pewne mentalne odgrodzenie koncepcji domenowych od koniecznych do spełnienia wymagań technicznych.

Praca z repozytoriami powinna koncepcyjnie przypominać pracę z kolekcjami [Evans, 2004]. Kod kliencki używający repozytorium powinien traktować je jak kolekcję obiektów w pamięci. Nie pozwala to jednak programistom piszących kod kliencki ignorować fakt, jak naprawdę dane repozytorium jest zaimplementowane. Taka ignorancja mogłaby mieć daleko idące negatywne skutki wydajnościowe, gdyby na przykład korzystając z repozytorium ładowano zbyt dużą liczbę obiektów do pamięci.

Kolejną cechą repozytoriów jest ich silne powiązanie z korzeniami agregatów. Repozytoria powinny operować jedynie na korzeniach agregatów, a nie na pojedynczych encjach czy obiektach wartości. Mogą służyć do zapisu jedynie korzeni agregatów, a przy odczycie mogą ewentualnie, poza zwracaniem korzeni, zapewniać informacje ogólne dot. agregatów, np. czy dany agregat istnieje lub ile jest agregatów spełniających dane warunki wyszukiwania.

7.1 Współpraca z innymi technologiami

Przy pracy z repozytoriami należy uważać, żeby nie skusić się na rozwiązania techniczne, które powodują wymieszanie abstrakcji warstwy domeny z abstrakcją utrwalania danych i dostępu do nich. Szczególnie należy zwrócić uwagę na transakcje i mapowanie obiektowo-relacyjne.

Udany zapis danych, który nie spowoduje braku spójności z innymi danymi jest możliwy dzięki wykorzystaniu mechanizmu transakcji. Łatwo jest wpaść w pułapkę umieszczenia kodu odpowiedzialnego za zarządzanie taką transakcją w implementacji repozytorium, które jest bardzo blisko zapisu danych. Takie podejście jest naiwne — repozytorium nie posiada odpowiedniego kontekstu do stwierdzenia, czy dana transakcja powinna zostać zatwierdzona czy nie. Repozytorium operuje na logice domenowej i nie ma, ani nie powinno mieć wiedzy zarezerwowanej logice aplikacyjnej, takiej jak to, kiedy rozpoczyna się i kończy transakcja w danym przypadku użycia aplikacji. Zarządzanie transakcjami na poziomie logiki aplikacyjnej jest zdecydowanie łatwiejsze, niż na poziomie jednego repozytorium. Nie zmienia to jednak faktu, że zawsze celem powinna być zmiana jednego agregatu w ramach jednej transakcji.

Mapowanie obiektowo-relacyjne (ang. ORM, object-relational mapping) i narzędzia do tego służące (np. Hibernate) stały się na tyle wygodne w użyciu, że bardzo łatwo i prosto wprowadza się je do projektów informatycznych. Użycie odpowiednich adnotacji w klasach mapowanych na tabele wydaje się mało inwazyjnym powiązaniem modelu obiektowego z modelem relacyjnym. Jest to jednak pomieszanie abstrakcji i złamanie zasady SRP [Martin, 2003, s. 95], gdyż tak zadnotowana klasa, oprócz swojej odpowiedzialności domenowej nabywa odpowiedzialność związaną z utrwalaniem jej stanu. Korzystanie z mapowania obiektowo-relacyjnego w warstwie domenowej ma jednak dużą liczbę zwolenników, głównie z powodu mniejszej ilości kodu do napisania i utrzymywania, niż gdyby stworzyć osobny model domeny i osobny model utrwalania stanu tej domeny.

Podejście z osobnym modelem utrwalania stanu nie doczekało się póki co większej uwagi, ale uważam, że temat będzie w przyszłości rozwijany. Tylko takie stanowcze odgrodzenie się od modelu domeny zapewnia jej odpowiednią hermetyzację potrzebną do realizacji głównego celu DDD — modelowaniu obiektów jak najbardziej odpowiadającym mentalnym obiektom biznesowym. Podobnie jak obecnie przyjęło się już, że do prezentacji obiektów domeny wykorzystuje się obiekty DTO m.in. z takiego powodu, że prezentowane dane mogą się różnić od modelowanych pojęć, tak wydaje się sensowne wprowadzenie podobnego rozwiązania przy utrwalaniu stanu obiektów domeny. Utrwalanie danych wykorzystujące chociażby mechanizm mapowania obiektowo-relacyjnego jest obarczone wieloma ograniczeniami, które nie po-

winny mieć wpływ na model domeny. Obiekty, które podlegałyby mapowaniu obiektowo-relacyjnym, a które przechowywałyby stan obiektów domenowych jednocześnie nie należąc do warstwy domeny nazwałbym w skrócie DBO (ang. database objects, pol. obiekty bazodanowe).

7.2 Odpytywanie repozytorium

Jednym z podstawowych zadań repozytorium jest zwracanie odpowiednich danych. Do implementacji zapytań wykorzystuje się dwa podejścia:

- konkretne metody wyszukujące;
- zapytania oparte o specyfikację [Evans i Fowler, 1997].

Konkretne metody wyszukujące to podstawowy i domyślny sposób implementacji zapytań w repozytorium. Dzięki użyciu takich konkretnie zdefiniowanych metod implementacja zapytania czy to do bazy danych, czy do innych systemów przechowujących dane, jest prosta i powtarzalna. Ponadto, ich użycie powoduje, że w aplikacji wiadomo, w jaki sposób klient odpytuje repozytorium. Daje to możliwość łatwiejszego dokonania optymalizacji zapytania lub wymiany jego implementacji. Wadą tego podejścia jest fakt, że nie jest ono bardzo elastyczne. Każdy dodatkowy atrybut wyszukiwania wymaga dodania dodatkowej metody do repozytorium. Na ten problem częściowo odpowiada mechanizm metod domyślnych w interfejsach wprowadzony do języka Java w wersji 8, który pozwala ograniczyć wpływ rozrastania się interfejsu na rozrastanie się implementacji. Duża liczba specyficznych metod służących do odczytów umieszczona w repozytorium może również sugerować, że nieumiejętnie zostało przeprowadzone oddzielenie stosów zapisu i odczytu [Fowler, 2011]. Repozytoria nie powinny zwracać danych służących jedynie do ich prezentacji przez warstwę odczytu.

Innym sposobem implementacji zapytań może być wykorzystanie obiektu specyfikacji. Obiekt specyfikacji jest formą predykatu i zamyka w sobie kryteria, według których należy wykonać zapytanie na repozytorium. To podejście powoduje, że repozytorium potrzebuje jedynie jednej metody, która pozwoli na jego odpytywanie za pomocą jakichkolwiek kryteriów umieszczonych w specyfikacji. Taki generyczny mechanizm jest trudniejszy do stworzenia implementując dane repozytorium; nie pozwala na łatwą optymalizację zapytania. Może też powodować przeciek abstrakcji związanej z wyszukiwaniem danych do warstwy domeny. Użycie tego podejścia może również sugerować nieumiejętne korzystanie z repozytorium, tak jak mnogość metod w poprzednim podejściu.

Odpowiednim domyślnym sposobem implementacji odpytywania repozytorium wydaje się sposób wykorzystujący konkretne metody wyszukujące. Jest łatwiejszy do wytworzenia i pozwala precyzyjnie korzystać z repozytorium minimalizując liczbę i poziom skomplikowania przekazywanych parametrów. Głównym problemem związanym z używaniem specyfikacji jest przeniesienie wiedzy w jaki sposób filtrować dane z warstwy implementacji repozytorium (warstwy infrastruktury) do warstwy domeny.

7.3 Implementacja

W kodzie aplikacji repozytorium składa się z dwóch części: interfejsu, który jest częścią warstwy domenowej oraz implementacji tego interfejsu, która jest częścią warstwy infrastruktury. Część infrastrukturalna jest zależna od wykorzystywanej technologii do utrwalania danych. Może to być np.: kolekcja obiektów w pamięci, baza danych, inna aplikacja, system plików, arkusz kalkulacyjny. W związku z mnogością technologii i bibliotek służących do ich obsługi nie da się stworzyć ogólnych zasad implementacji repozytorium po stronie infrastruktury. W związku z tym, analizie został poddany interfejs repozytorium.

Poniżej znajduje się przykład dobrego interfejsu repozytorium służącego do przechowywania agregatu `Product`:

Listing 7.1: Przykład wzorowego interfejsu repozytorium

```
interface ProductRepository {

  Optional<Product> findOne(ProductId productId);

  default Product getOne(ProductId productId) {
    return findOne(productId).orElseThrow(() ->
        new ProductNotFoundException(productId));
  }

  Optional<Product> findOneByName(String name);

  Set<Product> findByReleaseDateBefore(LocalDate date);

  Set<NotYetReleasedProduct> findNotYetReleased();

  long countByDiscountId(DiscountId discountId);

  boolean exists(ProductId productId);

  void save(Product product);
```

```
  void remove(ProductId productId);
}
```

Przykład przedstawia dziewięć metod będących przykładami głównych operacji repozytorium, które polegają na:

1. wyszukiwaniu korzenia agregatu na podstawie jego unikalnego ID;
2. zwracaniu korzenia agregatu na podstawie jego unikalnego ID;
3. wyszukiwaniu pojedynczego korzenia agregatu na podstawie innego unikalnego atrybutu;
4. wyszukiwaniu zbioru korzeni agregatów na podstawie nieunikalnego atrybutu (-ów);
5. wyszukiwaniu zbioru korzeni agregatów będących ich specjalnym rodzajem (podklasą);
6. zwracaniu danych podsumowujących, np. liczby korzeni agregatu;
7. zwracaniu informacji typu prawda-fałsz o istnieniu agregatu, spełniającego określone warunki;
8. zapisie agregatu (poprzez zapis jego korzenia);
9. usunięciu agregatu (poprzez usunięcie jego korzenia i zależnych obiektów — kaskadowo).

7.3.1 Konwencje nazewnicze

Konwencje nazewnicze metod wywodzą się z tych używanych w narzędziu Spring Data JPA, które umożliwia generację technicznej implementacji repozytoriów na podstawie odpowiednich nazw metod w interfejsie [Darimont et al., 2017]. Nazewnictwo wprowadzone przez Spring Data JPA jest spójne i warte rozprzestrzeniania. Na jego podstawie można zdefiniować reguły nazewnictwa metod:

- słowa `count` , `exists` , `save`, `remove` mówią same za siebie;
- wyszukiwanie pojedynczego elementu zawiera słówko `one`;
- wprowadzenie atrybutu po którym następuje wyszukiwanie poprzedza łącznik `by`;
- wyszukiwanie po ID nie wymaga łącznika `by` — podkreśla to fakt, że atrybut jest tożsamością agregatu;

- słowo `get` różni się od słowa `find` tym, że metoda zawierająca słowo `get` musi zwrócić korzeń agregatu (w przeciwnym wypadku jest to błąd programu), a metoda `find` — może zwrócić pusty obiekt `Optional` lub pusty zbiór.

Czasem można spotkać metody zawierające sufiks *-all*, np.: `findAll()`, `saveAll()` lub `removeAll()`. Takie metody nie powinny znajdować się w domenowym repozytorium. Ich użycie sugeruje, że modelowany agregat posiada swój korzeń „wyżej", niż obecnie. Takie metody łamią także zasadę modyfikacji jednego agregatu w ramach jednej transakcji.

7.3.2 Typy zwracane

Typy zwracane metod zamykają się w pięciu rodzajach:

1. pojedynczy korzeń agregatu (lub jego specyficzny rodzaj) — zwracany przez operację typu 2.;
2. opcjonalny korzeń agregatu (lub jego specyficzny rodzaj) — to opakowany typ 1. w typ rodzaju `Optional`, np. `java.util.Optional`; zwracany przez operację typu 1. i 3.;
3. zbiór korzeni agregatu (lub jego specyficznych rodzajów) — zwracany przez operację typu 4. i 5.;
4. meta-informacje o agregatach — mogą to być różne informacje związane z przechowywanymi agregatami; zwracane przez operację typu 6. i 7.;
5. `void`, gdy jest to wynik operacji zmieniającej stan repozytorium, czyli typu 8. lub 9.

Należy zwrócić uwagę, że zwracając więcej korzeni agregatu naraz powinno się użyć kolekcji typu `Set` (pol. zbiór). Ta zasada minimalizuje problemy związane ze zwracaniem obiektów w określonej kolejności. W ramach pracy w warstwie domenowej kolejność zwracanych z repozytorium obiektów nie powinna mieć żadnego znaczenia. Kolejność i związane z nią sortowanie należą do mechanizmów wykorzystywanych na potrzeby warstwy prezentacji i powinny znaleźć się w stosie odczytu (ang. Read, w CQRS). Repozytorium jest obiektem wykorzystywanym jedynie w stosie zapisu (ang. Write).

Operacje typu 2., zwracające typ rodzaju 1., czyli konkretny pojedynczy korzeń agregatu, rozpoczynające się słowem *get-* można w łatwy sposób zaimplementować używając mechanizmu wprowadzonego do języka Java w wersji 8 — domyślnych metod w interfejsie. Metody `getOne` korzystają

z implementacji metod `findOne` i do pełnej implementacji wystarczy dopisać do nich kawałek związany z rzuceniem odpowiedniego wyjątku.

7.3.3 Generyczna implementacja

Wszystkie interfejsy repozytoriów wyglądają podobnie. Z tego powodu istnieją fałszywe przesłanki do tworzenia tzw. generycznych repozytoriów, dla których wystarczy określić typ przechowywanego obiektu i cały mechanizm pozwoli na użycie tych samych metod w każdym z repozytoriów. Jest to jednak idea sprzeczna z przeznaczeniem repozytorium, jako domenową abstrakcję na przechowywanie danych.

Repozytorium istnieje w warstwie domenowej dlatego, że realizuje konkretne, a więc nie generyczne, operacje biznesowe. Operacje odpowiednie w jednym kontekście, dla jednego agregatu, nie będą odpowiednie w innym. Niektórych agregatów nie należy usuwać (są co najwyżej archiwizowane), inne nie są dodawane lub zmieniane przez system (działają w trybie tylko do odczytu) [Kaliszewski, 2016]. Przykładem nieodpowiedniej generycznej metody jest także metoda zwracająca wszystkie agregaty danego typu — zwykle jest ona tworzona w repozytorium mając na celu korzystanie z niej przy wyświetlaniu danych użytkownikowi. Takie zachowanie repozytorium jest niepoprawne, z przytaczanego już wcześniej powodu, że nie należy używać repozytoriów w stosie odczytu, a jedynie w stosie zapisu.

Dużo złych nawyków związanych z nieodpowiednim stosowaniem repozytorium i jego generycznej implementacji powoduje bardzo popularne narzędzie wspomniane powyżej — Spring Data. Zgodnie z tym narzędziem wystarczy stworzyć interfejs, który będzie rozszerzał generyczny interfejs o nazwie `org.springframework.data.repository.Repository` (dalej: `Repository`) lub jego podklasy. Same rozszerzenie interfejsu `Repository` nie jest jeszcze błędem, gdyż interfejs ten nie posiada żadnych metod, ale jest niebezpiecznym krokiem w złą stronę. Mechanizm Spring Data nie pozwoli np. na użycie dowolnych nazw metod — muszą one mieć odpowiednią strukturę, żeby można było dla nich wygenerować implementację, w związku z czym ogranicza to pole manewru przy modelowaniu dziedziny. Dodatkowo, bardzo łatwo da się stworzyć nieodpowiednie dla domenowego repozytorium metody `findAll()` lub `removeAll()`.

Narzędzie Spring Data jest bardzo przyjemne i efektywne w użyciu. Należy jednak pamiętać, że coś nazywające się „Repozytorium" nie musi być odpowiednim repozytorium w warstwie domenowej. Z powyższych względów rekomenduję użycie narzędzia Spring Data w warstwie infrastrukturalnej, tam gdzie znajduje się implementacja interfejsu repozytorium stworzonego jedynie z myślą o domenie, a nie zewnętrznych narzędziach.

7.3.4 Testowa podróbka implementacji

Konkretna implementacja interfejsu repozytorium, a więc jego część mająca miejsce w warstwie infrastruktury nie jest wzięta pod uwagę w tej pracy. Jednakże, do prezentacji jeszcze jednej zalety płynącej z zastosowania repozytorium, jako obiektu domenowego, przyczyni się zwrócenie uwagi na jeden specyficzny rodzaj konkretnej implementacji repozytorium — testowej podróbki implementacji.

Testując aplikację używa się różnych testowych sobowtórów (ang. test doubles) zamiast prawdziwych implementacji. Dotyczy to przede wszystkim warstw dostępu do danych, wywołań zdalnych procedur lub innych systemów. Istnieje wiele rodzajów tych sobowtórów, takich jak: podróbka (ang. fake), zaślepka (ang. stub) czy imitacja (ang. mock) [Fowler, 2007, Lipski, 2017].

Używając obiektu domenowego repozytorium można w prosty sposób użyć jego testowego sobowtóra w testach. Użycie w warstwie domeny repozytorium jedynie jako interfejsu pozwala na jej odpowiednie przetestowanie bez podmiany jakiejkolwiek klasy w tej warstwie. Podmiana implementacji następuje dopiero w warstwie infrastruktury. Najwłaściwszym rodzajem sobowtóra implementacji repozytorium jest jego podróbka. Nie wymaga ona żadnego zewnętrznego narzędzia. Należy stworzyć implementację repozytorium, która będzie przechowywała korzenie agregatu w kolekcji, w pamięci. Najwłaściwszym rodzajem kolekcji w tym wypadku jest mapa o kluczach będących tożsamościami korzeni agregatów (ich identyfikatorami) oraz wartościach stanowiących konkretne korzenie agregatów.

Do celów testowych świetnie sprawdzi się implementacja `HashMap`, gdyż testy zazwyczaj są uruchamiane w jednym wątku. W przypadku, gdy wiele wątków korzystałoby z danej testowej podróbki implementacji, należałoby użyć implementacji `ConcurrentHashMap`, gdyż zapewnia ona bezpieczeństwo danych przy użyciu wielowątkowym (kosztem szybkości w porównaniu do `HashMap`).

Fragment przykładowej testowej podróbki implementacji został zaprezentowany poniżej:

Listing 7.2: Fragment przykładowej testowej podróbki implementacji repozytorium

```
class ProductRepositoryTestFake
    implements ProductRepository {

  private final Map<ProductId, Product> products
      = new HashMap<>();

  @Override
  public Optional<Product> findOne(ProductId productId) {
    return Optional.ofNullable(
      products.get(productId)
    );
  }

  @Override
  public Set<Product> findByReleaseDateBefore(
      LocalDate date) {
    return products
      .values()
      .stream()
      .filter(product -> product.releaseDate()
                                .isBefore(date))
      .collect(Collectors.toSet());
  }

  @Override
  public boolean exists(ProductId productId) {
    return findOne(productId).isPresent();
  }

  @Override
  public void save(Product product) {
    products.put(product.id(), product);
  }

  @Override
  public void remove(ProductId productId) {
    products.remove(productId);
  }

  // ...
}
```

Rozdział 8

Usługi domenowe

Usługi domenowe (ang. Domain Services, inaczej: serwisy domenowe) to obiekty domenowe realizujące znaczący proces lub transformację w obrębie domeny, który nie jest naturalną odpowiedzialnością encji lub obiektu wartości [Evans, 2004]. Usługi domenowe mogą angażować wiele obiektów domenowych, w szczególności — wiele agregatów.

Istnieją pewne cechy kodu, które mogą świadczyć o tym, że daną metodę realizująca logikę biznesową należy wziąć pod uwagę jako kandydat na usługę domenową. Są to takie sytuacje, w których metoda:

- nie pasuje do żadnego konkretnego obiektu domenowego; nie operuje na żadnych polach; może być kojarzona z tzw. klasą użytkową (ang. utility class);
- wymaga do działania więcej niż jednego agregatu.

8.1 Klasy użytkowe

W przypadku pierwszej cechy, powszechnie uznawaną zasadą jest to, żeby nie tworzyć publicznych statycznych metod lub klas użytkowych. Utrudniają one testowanie i nie pozwalają na łatwe modyfikacje w przyszłości [Rasiński, 2016]. Metody, które są opisywane w ten sposób, a które są częścią domeny, zawsze powinny być modelowane jako usługi domenowe.

8.2 Kolaboracje agregatów

Druga cecha kodu jest dobrze wyczuwalna, gdy dwa (lub więcej) agregaty nie mają ze sobą relacji, np. w procesie autoryzacji, który wymaga niezależnych od siebie agregatów, np. użytkownik oraz metoda logowania.

Użytkownik nie posiada żadnych informacji o metodach logowania, a metoda logowania nie bierze pod uwagę użytkowników, jednak do poprawnej realizacji procesu autoryzacji musi zostać użyta właściwa metoda logowania oraz niezależnie — użytkownik musi poświadczyć swoją tożsamość.

Sytuacja komplikuje się, gdy jeden agregat jest zależny od drugiego, np. jeden zawiera referencję do drugiego. Rozważmy następującą sytuację: istnieje agregat `Product` zawierający referencję do agregatu `Discount` (w postaci `DiscountId`). Agregat `Product` zawiera także datę jego wydania, a agregat `Discount` — informację o wartości zniżki.

Listing 8.1: Fragment agregatu `Product`

```
class Product {

  private LocalDate releaseDate;
  private DiscountId discountId;

  // ...
}

class Discount {

  private DiscountValue value;

  // ...
}
```

Przykładowy przypadek użycia do zaimplementowania to zmiana daty wydania produktu na wcześniejszą. Jednakże trzeba wziąć pod uwagę regułę biznesową mówiącą, że datę wydania produktu można zmienić jedynie w przyszłości i gdy zmienia się ją na datę bliską (od teraz za co najwyżej 30 dni) to zniżka na ten produkt nie może być wyższa niż 10%.

8.2.1 Implementacja bez użycia usługi domenowej

Istnieje pokusa, żeby zamodelować zmianę daty wydania produktu wyłącznie w agregacie `Product`, który podczas tej zmiany sprawdzałby, czy będzie to data bliska i wtedy sprawdzałby, czy wartość zniżki jest odpowiednia. Poniżej przykładowa implementacja tego podejścia zaczynając od serwisu aplikacyjnego `ProductApplicationService`:

Listing 8.2: Przykład implementacji bez użycia usługi domenowej

```
public class ProductApplicationService {
  private final ProductRepository productRepository;
```

```
  private final DiscountRepository discountRepository;
  private final TimeProvider timeProvider;

  public ProductApplicationService(
      ProductRepository productRepository,
      DiscountRepository discountRepository,
      TimeProvider timeProvider) {
    this.productRepository = productRepository;
    this.discountRepository = discountRepository;
    this.timeProvider = timeProvider;
  }

  public void changeReleaseDate(ProductId productId,
                                LocalDate releaseDate) {
    Product product = productRepository.getOne(productId);
    product.changeReleaseDate(releaseDate, timeProvider,
        discountRepository);
    productRepository.save(product);
  }
}

class Product {
  private final ProductId id;
  private LocalDate releaseDate;
  private DiscountId discountId;

  private final static Period CLOSE_TO_RELEASE_PERIOD =
      Period.ofDays(30);

  private final static DiscountValue
      MAX_DISCOUNT_FOR_PRODUCT_CLOSE_TO_RELEASE =
      new DiscountValue(new BigDecimal("0.1"));

  Product(ProductId id) {
    this.id = id;
  }

  void changeReleaseDate(
      LocalDate newReleaseDate,
      TimeProvider timeProvider,
      DiscountRepository discountRepository) {
    assertNewReleaseDateInTheFuture(
        newReleaseDate, timeProvider);
    assertDiscountNotExceedsMaxForProductCloseToRelease(
        timeProvider, discountRepository);
    releaseDate = newReleaseDate;
  }

  private void assertNewReleaseDateInTheFuture(
```

```
      LocalDate newReleaseDate,
      TimeProvider timeProvider) {
    if (!newReleaseDate.isAfter(timeProvider.today())) {
      throw new NewReleaseDateNotInTheFuture(newReleaseDate);
    }
  }

  private void
      assertDiscountNotExceedsMaxForProductCloseToRelease(
          TimeProvider timeProvider,
          DiscountRepository discountRepository) {
    if (isNotCloseToRelease(timeProvider)) {
      return;
    }
    Discount discount =
        discountRepository.getOne(discountId);
    if (discount.isExceeding(
        MAX_DISCOUNT_FOR_PRODUCT_CLOSE_TO_RELEASE)) {
      throw new DiscountExceedsMaxForProductCloseToRelease(
          discount);
    }
  }

  private boolean isNotCloseToRelease(
      TimeProvider timeProvider) {
    return !isCloseToRelease(timeProvider);
  }

  private boolean isCloseToRelease(
      TimeProvider timeProvider) {
    return timeProvider.today()
                       .plus(CLOSE_TO_RELEASE_PERIOD)
                       .isAfter(releaseDate);
  }
}
```

Klasa `Product` odnosi się do agregatu `Discount` jedynie poprzez posiadany identyfikator, a nie poprzez jego referencję (zgodnie z regułą wprowadzoną w rozdziale 5 o agregatach), dlatego w metodzie zmieniającej datę wydania musi przyjąć repozytorium tych agregatów jako jeden z argumentów. Zaletą tego podejścia jest wykonanie całej logiki w jednym miejscu — agregacie — który utrzymuje swoje niezmienniki (chociaż nawet nie do końca zależne od niego). Największą wadą tego podejścia jest mała zrozumiałość i czytelność metody zmieniającej datę wydania. Podawanie repozytorium dopłat w metodzie, która zmienia jedynie datę wydania produktu jest niespodziewane, narusza „zasadę najmniejszego zaskoczenia” [Raymond, 2003]. Ponadto, takie repozytorium staje się zależnością podaną przez metodę, a więc

musi zostać w ten sposób podawane przez kolejne metody, aż do momentu jego użycia. Te podejście skomplikuje się też bardziej, gdy oprócz jednego repozytorium będą potrzebne inne.

Dodatkową kwestią, który należy rozważyć przy tym podejściu, jest umiejscowienie stałej reprezentującej maksymalną wartość dopłaty dla produktu bliskiego dacie wydania. Została ona umieszczona w klasie `Product`, ale prawie równie dobrze mogła by być umieszczona w klasie `Discount`. To zależałoby od zależności pomiędzy tymi agregatami. W przytoczonym przykładzie agregat `Product` jest świadomy agregatu `Discount` (operuje na nim), natomiast agregat `Discount` nie musi nic wiedzieć o agregacie `Product`. W momencie, gdy jednak ta relacja stała by się bilateralna to można by umiejscowić tą stałą w obojętnie której klasie — nie wiadomo było by, gdzie jej szukać i kto jest za nią odpowiedzialny.

8.2.2 Implementacja z usługą domenową

Lepszym rozwiązaniem jest hermetyzacja logiki związanej ze zmianą daty do dedykowanej klasy domenowej — usługi domenowej. Poniżej przykład implementacji z zastosowaniem tego rozwiązania:

Listing 8.3: Przykład implementacji z usługą domenową

```
public class ProductApplicationService {
  private final ProductReleaseDateChanger
      productReleaseDateChanger;

  public ProductApplicationService(
      ProductReleaseDateChanger productReleaseDateChanger) {
    this.productReleaseDateChanger =
        productReleaseDateChanger;
  }

  public void changeReleaseDate(ProductId productId,
                                LocalDate releaseDate) {
    productReleaseDateChanger.changeReleaseDate(productId,
        releaseDate);
  }
}

class ProductReleaseDateChanger {
  private final ProductRepository productRepository;
  private final DiscountRepository discountRepository;
  private final TimeProvider timeProvider;

  private final static DiscountValue
```

```
      MAX_DISCOUNT_FOR_PRODUCT_CLOSE_TO_RELEASE =
        new DiscountValue(new BigDecimal("0.1"));

  ProductReleaseDateChanger(
      ProductRepository productRepository,
      DiscountRepository discountRepository,
      TimeProvider timeProvider) {
    this.productRepository = productRepository;
    this.discountRepository = discountRepository;
    this.timeProvider = timeProvider;
  }

  void changeReleaseDate(ProductId productId,
                         LocalDate releaseDate) {
    Product product = productRepository.getOne(productId);

    assertNewReleaseDateInTheFuture(releaseDate);
    assertDiscountNotExceedsMaxForProductCloseToRelease(
        product);

    product.changeReleaseDate(releaseDate);

    productRepository.save(product);
  }

  private void assertNewReleaseDateInTheFuture(
      LocalDate newReleaseDate) {
    if (!newReleaseDate.isAfter(timeProvider.today())) {
      throw new NewReleaseDateNotInTheFuture(newReleaseDate);
    }
  }

  private void
      assertDiscountNotExceedsMaxForProductCloseToRelease(
          Product product) {
    if (product.isNotCloseToRelease(timeProvider.today())) {
      return;
    }
    Discount discount = discountRepository.getOne(
        product.discountId());
    if (discount.isExceeding(
        MAX_DISCOUNT_FOR_PRODUCT_CLOSE_TO_RELEASE)) {
      throw new DiscountExceedsMaxForProductCloseToRelease(
            discount);
    }
  }
}

class Product {
```

```
  private final ProductId id;
  private LocalDate releaseDate;
  private DiscountId discountId;

  private final static Period CLOSE_TO_RELEASE_PERIOD =
      Period.ofDays(30);

  Product(ProductId id) {
    this.id = id;
  }

  LocalDate releaseDate() {
    return releaseDate;
  }

  DiscountId discountId() {
    return discountId;
  }

  void changeReleaseDate(LocalDate newReleaseDate) {
    releaseDate = newReleaseDate;
  }

  boolean isNotCloseToRelease(LocalDate today) {
    return !isCloseToRelease(today);
  }

  private boolean isCloseToRelease(LocalDate today) {
    return today.plus(CLOSE_TO_RELEASE_PERIOD)
                .isAfter(releaseDate);
  }
}
```

Użycie dedykowanej usługi domenowej pozwala zawrzeć w jednym miejscu logikę walidacyjną angażującą więcej niż jeden agregat. Dzięki temu, w serwisie aplikacyjnym potrzeba mniej zależności (wystarcza jedna — dot. zmiany), podobnie jak w agregacie `Product`, który przy zmianie daty wydania nie potrzebuje żadnej dodatkowej informacji poza nową datą. Agregat `Product` nie musi dbać o utrzymywanie spójności niezmienników, które są niezależne od posiadanych przez niego informacji. Utrzymanie spójności zgodnie z regułami biznesowymi pozostaje jednak w warstwie domenowej, gdyż usługa domenowa znajduje się w tej warstwie.

Takie rozwiązanie powoduje mniejsze sprzężenie (ang. coupling), co jest pożądaną cechą kodu obiektowego. Każda z klas staje się łatwiej testowalna i bardziej czytelna, poprzez wyraźnie oddzielone odpowiedzialności, zgodnie z zasadą pojedynczej odpowiedzialność (ang. single responsibility principle

[Martin, 2003, s. 95]). Usługa domenowa odpowiada za całą logikę zmiany daty wydania produktu, która jest niezależna od samego produktu i jego niezmienników. To rozwiązuje problem lokalizacji stałej dot. maksymalnej wartości dopłaty przy zmianie.

Wadą takiego podejścia można uznać to, że jest możliwe wywołanie metody zmiany daty wydania produktu bezpośrednio z agregatu `Product`, bez wykonywania wcześniejszych walidacji. To niebezpieczeństwo minimalizuje się przez ustawienie metodzie `changeReleaseDate()` dostępu pakietowego. W obrębie jednego pakietu ryzyko związane z nieodpowiednim wywołaniem metody zmniejsza się, ale dalej występuje. Inną wadą jest udostępnienie poza klasę `Product` informacji o tym, czy produkt zbliża się do daty wydania, która jest wykorzystywana przez usługę `ProductReleaseDateChanger`. Jest to jednakże metoda dostępowa, która nie pozwala na modyfikację stanu agregatu `Product` z zewnątrz (zwraca jedynie zmienną typu `boolean`), więc wada ta nie jest istotna.

8.3 Dodatkowe uwagi

Dobrze zaimplementowana usługa domenowa powinna być bezstanowa [Evans, 2004]. Jedyne pola, które może posiadać, to te mające referencje do innych bezstanowych zależności w obrębie domeny, takich jak repozytoria, fabryki lub inne usługi domenowe.

Ponadto, istotne jest to, żeby usługa domenowa spełniała tylko jedną funkcję. Dzięki temu pozostanie niewielki i łatwy do zrozumienia.

Decyzja o stworzeniu i korzystaniu z usługi domenowej musi być dobrze przemyślana. Stosunkowo łatwo jest ulec pokusie tworzenia dużej liczby usług domenowych, które mogą wyjaławiać domenę, tzn. tworzyć anemiczny jej model. Nie należy jednak popadać również w drugą skrajność i nie korzystać z takich usług w ogóle, gdyż ich brak może spowodować powstanie w agregatach zbyt bogatej domeny (czasem zwaną „barokową"). Praca z takimi agregatami staje się wtedy dużo trudniejsza i czasochłonna.

Rozdział 9

Podsumowanie

Celem niniejszej pracy magisterskiej było szczegółowe opracowanie sposobów na projektowanie i implementację elementów składowych Domain-Driven Design w języku Java. Wszystkie elementy składowe mieszczące się w zakresie tej pracy zostały opracowane. Dla każdego z elementów składowych stworzyłem przykładową lub wzorową implementację z jej szczegółowym omówieniem. W związku z powyższym cel pracy został osiągnięty.

Praca jeszcze bardziej przybliża pojęcia elementów składowych w DDD. W analizie poszczególnych elementów schodzi na niższy poziom abstrakcji niż ogólnodostępne i uznane źródła, dzięki czemu pozwala dokładniej zrozumieć niektóre koncepcje.

9.1 Wnioski

W ramach pracy, w każdym rozdziale omawiającym dany element składowy znalazła się co najmniej jedna koncepcja twórcza. Są to nowe wnioski lub proponowane nowe zasady powstałe głównie z analizy i kompilacji reguł wprowadzonych przez autorów podejścia DDD. W kolejnych sekcjach wymieniam je w skróconej formie.

9.1.1 Encje

Uznałem za najwłaściwszy sposób tworzenia identyfikatorów encji — ich autogenerację przez aplikację i wykorzystanie do tego UUID. Poleciłem opakowywanie każdego identyfikatora encji w charakterystyczny dla encji obiekt wartości. Zaproponowałem użycie prefiksu *change-* zamiast *set-* przy nazywaniu metod implementujących przypisanie nowej wartości pola klasy. Wyodrębniłem cztery rodzaje walidacji obiektu domenowego. Zarekomendowałem

porzucenie prefiksu *get-* przy tworzeniu metod dostępowych, jak i ostateczne zarzucenie implementacji standardu JavaBeans. Uznałem zwracanie struktur DTO, jako właściwe podejście do zastępowania licznych metod dostępowych.

9.1.2 Obiekty wartości

Zachęciłem, by opakowywać kolekcje obiektów w obiekty wartości. Wymieniłem dwa przypadki powstawania wielu obiektów wartości podobnego typu i zaproponowałem regułę, w jaki sposób można je rozróżniać. Stworzyłem implementacje referencyjne klas niemutowalnych i metod ich porównujących w języku Java, które można traktować jako gotowy wzór do zastosowania przy tworzeniu konkretnych implementacji takich klas i metod. W przypadku problemów ze skorzystaniem z wbudowanej metody `equals()` poleciłem stworzenie metody dedykowanej porównywaniu obiektów w domenie — metody `sameAs()`.

9.1.3 Agregaty

Wprowadziłem regułę tworzenia klas lokalnych agregatu z domyślnym dostępem pakietowym. Uzupełniłem charakterystykę agregatu o zasadę, zgodnie z którą agregat powinien zarządzać cyklem życia swoich obiektów lokalnych. Podkreśliłem wagę reguły odnoszenia się do innych agregatów jedynie poprzez ID ich korzenia.

9.1.4 Fabryki

Przedstawiłem cztery rodzaje fabryk ze względu na ich miejsce występowania. Skrytykowałem i wezwałem do zaprzestania używania jednego z rodzajów fabryk — metody wytwórczej w korzeniu innego agregatu. Zamiast niej, w każdym przypadku, w którym oryginalnie była sugerowana — polecam użycie dedykowanej klasy fabryki.

9.1.5 Repozytoria

Zasugerowałem za najwłaściwsze podejście do utrwalania danych w aplikacji podejście z osobnym od domeny modelem utrwalania stanu. Uznałem za odpowiedni domyślny sposób implementacji odpytywania repozytorium sposób wykorzystujący konkretne metody wyszukujące zamiast obiektu specyfikacji. Wyszczególniłem główne rodzaje metod występujących w repozytoriach. Ugruntowałem konwencje nazewnicze metod. Wyznaczyłem ograni-

czoną liczbę typów zwracanych przez metody. Uznałem za niewłaściwe korzystanie w warstwie domenowej z generycznych repozytoriów.

9.1.6 Usługi domenowe

Potępiłem użycie klas użytkowych i w zamian za nie, zasugerowałem tworzenie usług domenowych. Zaproponowałem używanie usług domenowych zawsze wtedy, gdy należy wykonać operacje oddziałujące na więcej niż jeden agregat. Przy braku stosowania usług domenowych wystosowałem ostrzeżenie o możliwości przeładowania agregatów słabo związanymi z nimi odpowiedzialnościami.

9.2 Perspektywy rozwoju pracy

Każdy element składowy DDD poruszany w niniejszej pracy ma różne, często osobliwe, przypadki jego zastosowania. W związku z tym, każdy z nich z osobna mógłby być analizowany w odrębnej pracy dyplomowej. Byłoby to niewątpliwie zejście niżej na kolejny poziom abstrakcji, co byłoby logiczną kontynuacją prac związaną z elementami składowymi DDD. Byłby to rozwój pracy wgłąb.

Dodatkowo, istnieją w społeczności DDD poglądy kategoryzujące jako elementy składowe takie elementy jak zdarzenia domenowe, specyfikacje czy polityki. Nie były one poruszane w niniejszej pracy, więc ta mogłaby zostać o nie powiększona. To mogłoby zostać uznane za rozwój pracy wszerz.

Ponadto wszystko, samo zagadnienie Domain-Driven Design jest na tyle szerokie, że można się zastanowić nad rozwojem pracy w innym jego dziale. Elementy składowe DDD są często nazywane wzorcami taktycznymi. Można więc, w nawiązaniu do nich, poddać podobnej analizie np. tzw. wzorce strategiczne.

Bibliografia

Bloch, Joshua, 2008. *Effective Java*. Wyd. II. Upper Saddle River, NJ: Addison-Wesley.

Charlton, Casey, 2009. *Domain Driven Design Step by Step Guide*. [e-book]. Dostępny przez: `https://pl.scribd.com/document/90519910/Domain-Driven-Design-Step-by-Step` [dostęp: 4 grudnia 2017].

Citerus AB. *DDDSample*. [projekt]. Dostępny w: `https://github.com/citerus/dddsample-core` [dostęp: 4 grudnia 2017].

Constantine, L., Myers, G. i Stevens, W., 1974. Structured Design. *IBM Systems Journal*, 13 (2), s. 115–139.

Darimont, Thomas, Gierke, Oliver, Paluch, Mark i Strobl, Christoph, 2017. Spring Data JPA - Reference Documentation. *Spring*, [dokumentacja online] (Ostatnia aktualizacja: 27 listopada 2017). Dostępna w: `https://docs.spring.io/spring-data/data-jpa/docs/2.0.2.RELEASE/reference/html/` [dostęp: 4 grudnia 2017].

Evans, Eric i Fowler, Martin, 1997. *Specifications*. [pdf]. Dostępny w: `https://www.martinfowler.com/apsupp/spec.pdf` [dostęp: 4 grudnia 2017].

Evans, Eric, 2004. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Boston, MA: Addison-Wesley.

Evans, Eric, 2007. *What is Domain-Driven Design?* Dostępny w: `http://dddcommunity.org/learning-ddd/what_is_ddd/` [dostęp: 4 grudnia 2017].

Ford, Tim, 2014. Clustered Indexes Based Upon GUIDs. *IT Pro*, 8 września. Dostępny w: `http://www.itprotoday.com/database-performance-tuning/clustered-indexes-based-upon-guids` [dostęp: 4 grudnia 2017].

Fowler, Martin, MacKenzie, Josh, Parsons, Rebecca, 2000. POJO. *martinfowler.com*, [blog] wrzesień.
Dostępny w: `https://www.martinfowler.com/bliki/POJO.html` [dostęp: 4 grudnia 2017].

Fowler, Martin 2003a. *Patterns of Enterprise Application Architecture*. Boston, MA: Addison-Wesley, s. 401–413.

Fowler, Martin, 2003b. AnemicDomainModel. *martinfowler.com*, [blog] 25 listopada.
Dostępny w: `https://www.martinfowler.com/bliki/AnemicDomainModel.html` [dostęp: 4 grudnia 2017].

Fowler, Martin, 2007. Mocks Aren't Stubs. *martinfowler.com*, [blog] (Ostatnia aktualizacja: 2 stycznia 2007).
Dostępny w: `https://martinfowler.com/articles/mocksArentStubs.html` [dostęp: 4 grudnia 2017].

Fowler, Martin, 2011. CQRS. *martinfowler.com*, [blog] 14 lipca.
Dostępny w: `https://martinfowler.com/bliki/CQRS.html` [dostęp: 4 grudnia 2017].

Gamma, Erich, Helm, Richard, Johnson, Ralph i Vlissides, John, 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley.

Guava: Google Core Libraries for Java, 2017. Wersja 23.5. [biblioteka].
Dostępna w: `https://github.com/google/guava` [dostęp: 4 grudnia 2017].

Hamilton, Graham, red., 1997. *Specyfikacja JavaBeans™ API*. Wersja 1.01-A. Mountain View, CA: Sun Microsystems.
Dostępna przez: `http://www.oracle.com/technetwork/java/javase/documentation/spec-136004.html` [dostęp: 4 grudnia 2017].

Holland, Ian, 1987. *Law of Demeter: Principle of Least Knowledge*.
Dostępny w: `http://www.ccs.neu.edu/home/lieber/LoD.html` [dostęp: 4 grudnia 2017].

Kaliszewski, Dawid, 2016. Repozytorium - najbardziej niepotrzebny wzorzec projektowy. *Commit... and run!*, [blog] 11 maja.
Dostępny w: `http://commitandrun.pl/2016/05/11/Repozytorium_najbardziej_niepotrzebny_wzorzec_projektowy/` [dostęp: 4 grudnia 2017].

Kay, Alan C., 1993. The early history of Smalltalk. *ACM SIGPLAN Notices*, tom 28, nr 3, s. 69–95.

Leach, P., Mealling, M. i Salz, R., 2005. *A Universally Unique IDentifier (UUID) URN Namespace*. RFC 4122.
Dostępny w: `https://www.ietf.org/rfc/rfc4122.txt` [dostęp: 4 grudnia 2017].

Lerman, Julie i Smith, Steve, 2014. Domain-Driven Design Fundamentals. *Pluralsight*, [kurs online] (Ostatnia aktualizacja: 25 czerwca 2014).
Dostępny w: `https://www.pluralsight.com/courses/domain-driven-design-fundamentals` [dostęp: 4 grudnia 2017].

Lipski, Michał, 2017. Test Doubles — Fakes, Mocks and Stubs. *Pragmatists blog*, [blog] 30 marca.
Dostępny w: `https://blog.pragmatists.com/test-doubles-fakes-mocks-and-stubs-1a7491dfa3da` [dostęp: 4 grudnia 2017].

Martin, Robert C., 2003. *Agile Software Development, Principles, Patterns, and Practices*. Upper Saddle River, NJ: Prentice Hall.

Meyer, Bertrand, 1988. *Object-Oriented Software Construction*. Upper Saddle River, NJ: Prentice Hall.

Nabrdalik, Jakub, 2017. *Keep IT clean*, [prezentacja].
Dostępna w: `https://jakubn.gitlab.io/keepitclean/` [dostęp: 4 grudnia 2017].

Nilsson, Jimmy, 2002. The Cost of GUIDs as Primary Keys. *InformIT*, [online] 8 marca.
Dostępny w: `http://www.informit.com/articles/printerfriendly/25862` [dostęp: 4 grudnia 2017].

PostgreSQL, 2017. *Documentation: 10: 8.1. Numeric Types*.
Dostępny w: `https://www.postgresql.org/docs/10/static/datatype-numeric.html` [dostęp: 4 grudnia 2017].

Rasiński, Mateusz, 2016. Do not create public static methods. *SoftwArt Blog*, [blog] 12 listopada.
Dostępny w: `https://mateuszrasinski.github.io/do-not-create-public-static-methods/` [dostęp: 4 grudnia 2017].

Raymond, Eric S., 2003. *The Art of UNIX Programming*. Boston, MA: Addison-Wesley, s. 254-255

Sobótka, Sławomir, 2008. UP-DDD in Action: Hermetyczne agregaty. *Holistycznie o inżynierii oprogramowania*, [blog] 14 grudnia. Dostępny w: `http://art-of-software.blogspot.com/2008/12/up-ddd-in-action-hermetyczne-agregaty.html` [dostęp: 4 grudnia 2017].

Sobótka, Sławomir, 2011. Domain Driven Design. *Software Developer's Journal*, nr 08/2011. Dostępny w: `https://bottega.com.pl/pdf/materialy/sdj-ddd.pdf` [dostęp: 4 grudnia 2017].

Sobótka, Sławomir, 2012. Domain Driven Design krok po kroku. Część I: Podstawowe Building Blocks DDD. *Programista*, nr 1/2012 (1), s. 38–46. Dostępny w: `https://programistamag.pl/wp-content/uploads/downloads/Programista_0_2012.pdf` [dostęp: 4 grudnia 2017].

Sobótka, Sławomir, 2013. *DDD-CqRS sample v2.0*. [projekt]. Dostępny w: `https://github.com/BottegaIT/ddd-leaven-v2` [dostęp: 4 grudnia 2017].

Stec-Fus, Dorota i Warszawski, Marcin, 2013. Kosztowny błąd PESEL-u. *Dziennik Polski*, [online] 17 grudnia. Dostępny w: `http://www.dziennikpolski24.pl/artykul/3293958,kosztowny-blad-peselu,id,t.html` [dostęp: 4 grudnia 2017].

Ustawa z dnia 24 września 2010 r. o ewidencji ludności, Art. 19. Dz.U. 2010 nr 217, poz. 1427.

Vernon, Vaughn, 2013. *Implementing Domain-Driven Design*. Boston, MA: Addison-Wesley.